GeekBrains

Создание Интернет-магазина, построенного по принципам Single Page Application архитектуры с использованием Фреймворка Vue 3 версии

Специальность:

Frontend-программист. Цифровые профессии.

Леншмидт Андрей Игоревич

Краснодар

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На сегодняшний день web-приложения уже стали неотъемлемой частью жизни почти каждого человека. Сначала это были лишь сайты-одностранички, сделанные на простом HTML и CSS, которые просто давали доступ к некоторой важной для людей информации, как книги в библиотеке или бумажные газеты. Затем появился JavaScript и сайты стали приобретать черты web-приложений, они стали интерактивными и дали обычному пользователю возможность более сложного взаимодействия с ними. На этом развитие web-технологий не остановилось, появились крупные библиотеки и фреймворки значительно ускорившие сложный и трудоемкий процесс разработки приложений. И уже сегодня практически каждый у кого есть доступ в интернет способен не только получить любую необходимую информацию в интернете, но также может воспользоваться услугами банков, записаться на прием в поликлинику, заказать еду с доставкой на дом, приобрести все необходимое в любимом Интернет-магазине.

Помогать пользователю комфортно и безопасно взаимодействовать с такими крупными и сложными web-сервисами должны пользовательские интерфейсы разработкой которых занимаются Frontend-разработчики. Важность данной работы трудно переоценить она позволит не только овладеть базовыми навыками разработки, но и создать полноценное Single Page Application (SPA в дальнейшем).

Тема проекта: Создание Интернет-магазина, построенного по принципам Single Page Application архитектуры с использованием Фреймворка Vue 3 версии.

Цель: Изучение фреймворка Vue 3 версии и создание полноценного SPA интернет-магазин с его помощью.

Дипломный проект будет разрабатываться по специализации Frontend-программист. На этапе создания данного проекта из опыта разработки SPA у меня за плечами имеется только один учебный проект в рамках курса "Фреймворк Vue.js" выполненный на образовательной платформе GeekBrains,

есть опыт разработки многостраничных приложений Multi Page Application (MPA в дальнейшем) с использованием препроцессора SCSS, выполненных также на платформе GeekBrains.

Суть проекта состоит в том, чтобы создать Frontend-часть интернет-магазина с возможностями получать данные о товарах с сервера, сохранять их в избранное, добавлять в корзину, с возможностью сортировать товары по категориям, и также сохранять данные пользователя локально.

Задачи проекта:

1. Изучить документацию по Vue 3 версии;
2. Разработать дизайн и структуру сайта;
3. Сделать асинхронные методы получения товаров с сервера;
4. Создать глобальное хранилище данных с помощью Vuex-store;
5. Настроить переходы по страницам с помощью Vue-router;
6. Сделать главную страницу приложения;
7. Сделать страницы каталога и избранных товаров;
8. Сделать корзину товаров;
9. Реализовать локальное хранение данных;
10. Сделать страницы регистрации, входа и личный кабинет пользователя;

Инструменты: редактор VScode, Vue Cli v5.0.8, Google DevTools, Node JS v21.6.2, Vue-router v4.3.0, Vuex v4.0.0, SCSS, Swiper v11.0.7,

На проекте участвовал в роли:

1. Архитектора приложения;
2. UX/UI Дизайнера;
3. Верстальщика;
4. Frontend-разработчика;
5. Тестировщика.

ГЛАВА 1
Знакомство с основными принципами и технологиями необходимыми для разработки web-приложения

1.1 Архитектуры web-приложений. Их преимущества и недостатки.

Архитектура веб-приложений – это способ организации и структурирования программного кода, который обеспечивает работу веб-приложения. Это как фундамент, на котором строится приложение.

Веб-архитектура включает в себя различные компоненты, такие как клиентская и серверная части, база данных и интерфейсы для взаимодействия с пользователем.

Можно выделить несколько типов архитектуры веб-приложений, в зависимости от того, как логика приложения распределяется между клиентской и серверной сторонами.

Есть несколько классификаций архитектур по типам. Из наиболее распространенных можно выделить следующие виды архитектуры веб-приложений:

1. Одностраничные веб-приложения (SPA);
2. Многостраничные веб-приложения (MPA);
3. Прогрессивные веб-приложения (PWA).

Они выделяются среди других подходов простотой разработки, удобством для пользователей и широкими возможностями для развития бизнеса.

Single Page Application (SPA) – веб приложение, представляющее из себя одну единственную HTML-страницу. В отличии от статичного сайта, здесь используется динамическое обновление, которое позволяет избежать загрузки других страниц. Это означает, что весь необходимый контент помещается в окне браузера. При переходе на другие страницы или разделы нужные элементы подгружаются без перезагрузки.

Работа с такими ресурсами напоминает взаимодействие не с веб-сайтами, а с полноценными десктопными приложениями. Такой эффект достигается

благодаря моментальному отклику на действия пользователя. Технически это стало возможно после появления фреймворков для языка JavaScript: React, Vue, Angular и других. С их помощью написаны такие популярные динамические приложения как: Facebook, Gmail, GitHub и Google Maps.

Одностраничные приложения имеют следующие достоинства:

1. Простота разработки – для визуального отображения страницы не нужно писать код для рендера на сервере;
2. Гибкость пользовательского интерфейса – благодаря тому, что приложение состоит всего из одной страницы, гораздо проще создать качественный интерфейс, наполненный множеством элементов. Кроме этого упрощаются процессы хранения сведений о сеансе и управлении анимацией;
3. Высокая скорость и экономия времени – чтобы загрузить все необходимые данные, нужна всего одна сессия, впоследствии недостающие элементы будут быстро подгружаться;
4. Возможность работать в offline-режиме – для сбора данных приложению достаточно одного запроса, после чего оно будет работать при помощи кэша.

Несмотря на все свои преимущества, у SPA есть и слабые стороны:

1. Проблемы с SEO оптимизацией – у таких приложений ограничено семантическое ядро, возникают сложности при анализе посещаемости, а динамический контент хуже индексируется поисковиками;
2. Поддержка JavaScript – без нее полноценно пользоваться приложением не получится, поскольку большинство функций просто не будут работать;
3. Поддержка браузеров – используемые при разработке фреймворки значительно повышают нагрузку на браузер, а также не всегда совместимы с ним;
4. Проблемы с безопасностью – поскольку в JavaScript бывают утечки памяти, такие ресурсы гораздо чаще подвергаются хакерским атакам.

Большинство перечисленных недостатков можно нивелировать путем использования специальных технологий разработки.

Multi Page Application (MPA) – это многостраничные приложения, которые работают по традиционной схеме. Это означает, что при каждом незначительном изменении данных или загрузке новой информации страница обновляется. Такие приложения тяжелее, чем одностраничные, поэтому их использование целесообразно только в тех случаях, когда нужно отобразить большое количество контента.

Многостраничные приложения более популярны в Интернете, так как в прошлом все веб-сайты были MPA. В наши дни компании выбирают MPA, если их веб-сайт довольно большой (например, eBay). Такие решения перезагружают веб-страницу для загрузки или отправки информации с/на сервер через браузеры пользователей.

Преимущества многостраничных приложений:

1. Простая SEO оптимизация – можно оптимизировать каждую из страниц приложения под нужные ключевые запросы;
2. Привычность для пользователей – за счет простого интерфейса и классической навигации.

К недостаткам относится:

1. Тесная связь между бекендом и фронтендом, поэтому их не получается развивать параллельно; сложная разработка – требуют использования фреймворков как на стороне клиента, так и на стороне сервера, что увеличивает сроки и бюджет разработки.
2. Сложная разработка – требуют использования фреймворков как на стороне клиента, так и на стороне сервера, что увеличивает сроки и бюджет разработки.

Progressive Web Application (PWA) – это веб-сайты, которые трансформируются в приложения. Они активно взаимодействуют с пользователем, например, отправляют push-уведомления, работают в offline-режиме и даже может работать с аппаратной частью смартфона (микрофоном,

камерой или геолокацией). Если приложению требуется доступ к функциям устройств, разработчики используют дополнительные API - NFC API, API геолокации, Bluetooth API и другие

К преимуществам прогрессивных веб-приложений относят:

1. Быструю разработку – для создания PWA достаточно внести изменения в уже существующий сайт;
2. Кроссплатформенность – приложение может работать на устройствах под управлением разных операционных систем;
3. Высокую скорость – PWA работают намного быстрее классических сайтов.

Существенным недостатком является то, что не все браузеры поддерживают основные функции таких приложений (например, Firefox и Edge)

По типу рендеринга можно выделить 2 основных виды архитектур:

1. - Рендеринг на стороне пользователя (CSR или Client Side Rendering) здесь вся работа по рендерингу приложения выполняется на стороне клиента, в браузере;
2. Рендеринг на стороне сервера (SSR или Server Side Rendering), способ генерации html на стороне сервера. В случае серверного рендеринга, после запроса клиентом странички, сервер на своей стороне выполняет API-запросы, а затем формирует html-страницу.

Данный тип классификации не является полностью независимым, а скорее выступает подтипом для всех архитектур в целом, и используется в комбинации вместе с ними. Например, связка SPA и SSR позволяет устранить некоторые характерные для SPA недостатки. К примеру, появляется возможность повысить эффективность поиска вашего сайта пользователями через поисковики, да и в целом приложение начинает работать еще быстрее, т.к. значительную часть работы выполняет мощный сервер, а на стороне пользователя тратится меньше ресурсов.

По структуре веб-приложений также можно выделить следующие виды архитектур:

1. Монолитное приложение;
2. Архитектура микросервисов;
3. Бессерверная архитектура;

Традиционная монолитная архитектура веб-приложения состоит из трех частей – базы данных, клиентской и серверной сторон. Это означает, что внутренняя и внешняя логика, как и другие фоновые задачи, генерируются в одной кодовой базе. Чтобы изменить или обновить компонент приложения, разработчики программного обеспечения должны переписать все приложение.

Что касается микросервисов, этот подход позволяет разработчикам создавать веб-приложение из набора небольших сервисов. Разработчики создают и развертывают каждый компонент отдельно.

Архитектура микросервисов выгодна для больших и сложных проектов, поскольку каждый сервис может быть изменен без ущерба для других блоков. Поэтому, если вам нужно обновить логику оплаты, вам не придется на время останавливать работу сайта.

Бессерверная архитектура веб-приложений заставляет разработчиков использовать облачную инфраструктуру сторонних поставщиков услуг, таких как Amazon и Microsoft.

Чтобы сохранить веб-приложение в Интернете, разработчики должны управлять серверной инфраструктурой (виртуальной или физической), операционной системой и другими процессами хостинга, связанными с сервером. Поставщики облачных услуг, такие как Amazon или Microsoft, предлагают виртуальные серверы, которые динамически управляют распределением машинных ресурсов. Другими словами, если ваше приложение испытывает огромный всплеск трафика, к которому ваши серверы не готовы, приложение не будет отключено.

В итоге правильный выбор архитектуры веб-приложения делает процесс разработки более эффективным и простым. Веб-приложение с продуманной архитектурой легче масштабировать, изменять, тестировать и отлаживать.

1.2 Три кита Frontend-разработки web-приложений HTML, CSS, JS.

Прежде чем рассматривать самые современные фреймворки, на которых сейчас разрабатывается практически весь Frontend стоит упомянуть три самых важных, базовых технологии HTML5, CSS3, JavaScript. Они как три кита в мире Frontend-разработки без которых работа современных приложений невозможна.

Первым и самым древним "технологическим китом" можно считать HTML.

HTML (HyperText Markup Language - язык гипертекстовой разметки) не является языком программирования; это язык разметки, используемый для определения структуры веб-страниц, посещаемых пользователями. Они могут иметь сложную или простую структуру, всё зависит от замысла и желания веб-разработчика. HTML состоит из ряда элементов, которые вы используете для того, чтобы охватить, обернуть или разметить различные части содержимого, чтобы оно имело определённый вид или срабатывало определённым способом. Встроенные тэги могут преобразовать часть содержимого в гиперссылку, по которой можно перейти на другую веб-страницу, выделить курсивом слова и так далее.

HTML является "скелетом" сайта его каркасом, определяющим структуру отображаемого документа. Считывая HTML-код браузер как по кирпичикам конструирует страницу документа. Базовым строительным блоком разметки является HTML-элемент.

Элемент, который содержит в себе контент состоит из следующих частей:

1. Открывающего тега (Opening tag): Состоит из имени элемента, заключённого в открывающие и закрывающие угловые скобки. Открывающий тег указывает, где элемент начинается или начинает действовать, в данном случае – где начинается абзац. Внутри тега могут быть указаны атрибуты элемента.

Атрибуты содержат дополнительную информацию об элементе, которую вы не хотите показывать в фактическом контенте. Например, атрибут *"class"* позволяет дать элементу идентификационное имя, которое может позже

использоваться, чтобы обращаться к элементу с информацией о стиле и прочих вещах.

Атрибут всегда должен иметь:

- пробел между ним и именем элемента (или предыдущим атрибутом, если элемент уже имеет один или несколько атрибутов);

- имя атрибута, за которым следует знак равенства;

- значение атрибута, заключённое с двух сторон в кавычки.

2. Закрывающего тега (Closing tag): Это то же самое, что и открывающий тег, за исключением того, что он включает в себя косую черту перед именем элемента. Закрывающий элемент указывает, где элемент заканчивается.

3. Контента (Content): Это контент элемента, который может является обычным текстом.

В свою очередь элементы, которые не содержат контента называются пустыми. Возьмём для примера элемент *<img>*

**

Он содержит два атрибута, но не имеет закрывающего тега *</img>*, и никакого внутреннего контента. Это потому, что элемент изображения не оборачивает контент для влияния на него. Его целью является вставка изображения в HTML страницу в нужном месте.

Структурно каждая HTML страница содержит в себе следующие теги:

<!DOCTYPE html> – доктайп. В прошлом, когда HTML был молод (около 1991/1992), доктайпы должны были выступать в качестве ссылки на набор правил, которым HTML страница должна была следовать, чтобы считаться хорошим HTML, что могло означать автоматическую проверку ошибок и другие полезные вещи. Однако в наши дни, никто не заботится об этом, и они на самом деле просто исторический артефакт, который должен быть включён для того, чтобы все работало правильно.

<html></html> – элемент *html*. Этот элемент оборачивает весь контент на всей странице, и иногда известен как корневой элемент.

<head></head> – элемент *head*. Этот элемент выступает в качестве контейнера для всего, что вы пожелаете включить на HTML страницу, но не являющегося контентом, который вы показываете пользователям вашей страницы. К ним относятся такие вещи, как ключевые слова и описание страницы, которые будут появляться в результатах поиска, CSS стили нашего контента, кодировка и многое другое.

<body></body> – элемент *body*. В нем содержится весь контент, который вы хотите показывать пользователям, когда они посещают вашу страницу, будь то текст, изображения, видео, игры, проигрываемые аудиодорожки или что-то ещё.

<meta charset="utf-8"> – это элемент, устанавливающий UTF-8 кодировку вашего документа, которая включает в себя большинство символов из всех известных человечеству языков. По сути, теперь документ может обрабатывать любой текстовый контент, который вы в него вложите. Нет причин не устанавливать её, так как это может помочь избежать некоторых проблем в дальнейшем.

<title></title> – элемент *title*. Этот элемент устанавливает заголовок для вашей страницы, который является названием, появляющимся на вкладке браузера загружаемой страницы, и используется для описания страницы, когда вы добавляете её в закладки/избранное.

Следующим за HTML технологическим китом можно считать CSS.

CSS (каскадные таблицы стилей) используется для стилизации и компоновки веб-страниц – например, для изменения шрифта, цвета, размера и интервала содержимого, разделения его на несколько столбцов или добавления анимации и других декоративных элементов.

CSS – это таблицы состоящая из набора правил, определяющих группы стилей, которые должны применяться к определённым элементам или группам

элементов на вашей веб-странице. Каждый из таких наборов правил заключен между фигурных скобок, перед которыми указывается селектор.

CSS-селектор – это часть CSS-правила, которая позволяет вам указать, к какому элементу (элементам) применить стиль. Сам селектор включая набор правил выглядит следующим образом:

```
h1 {
    color: red;
    font-size: 5em;
}
```

Здесь h1 – это селектор, а между фигурных скобок указаны сами правила, их также называют объявлениями. Они принимают форму пары свойства и его значения разделенными двоеточием, каждое правило отдаляется также точкой с запятой.

Ключевыми моментами CSS являются Каскад, Специфичность и Наследование.

Каскад таблицы стилей, если говорить упрощённо, означает, что порядок следования правил в CSS имеет значение; когда применимы два правила, имеющие одинаковую специфичность, используется то, которое идёт в CSS последним.

Специфичность определяет, как браузер решает, какое именно правило применяется в случае, когда несколько правил имеют разные селекторы, но, тем не менее, могут быть применены к одному и тому же элементу.

Степень специфичности, которой обладает селектор, измеряется с использованием четырёх различных значений (или компонентов), которые можно представить, как тысячи, сотни, десятки и единицы – четыре однозначные цифры в четырёх столбцах:

1. Тысячи: поставьте единицу в эту колонку, если объявление стиля находится внутри атрибута style (встроенные стили). Такие объявления не имеют селекторов, поэтому их специфичность всегда просто 1000.

2. Сотни: поставьте единицу в эту колонку за каждый селектор ID, содержащийся в общем селекторе.

3. Десятки: поставьте единицу в эту колонку за каждый селектор класса, селектор атрибута или псевдокласс, содержащийся в общем селекторе.

4. Единицы: поставьте общее число единиц в эту колонку за каждый селектор элемента или псевдоэлемент, содержащийся в общем селекторе.

5. Также существует особое правило "!important" которое используется, чтобы сделать конкретное свойство и значение самыми специфичными, таким образом переопределяя нормальные правила каскада.

Наследование также надо понимать в этом контексте – некоторые значения свойства CSS, установленные для родительских элементов наследуются их дочерними элементами, а некоторые нет.

CSS предоставляет четыре специальных универсальных значения свойства для контроля наследования. Каждое свойство CSS принимает эти значения:

1. *inherit* – устанавливает значение свойства, применённого к элементу, таким же, как у его родительского элемента. Фактически, это "включает наследование".

2. *initial* – устанавливает значение свойства, применённого к выбранному элементу, равным *initial value* этого свойства (в соответствии с настройками браузера по умолчанию. Если в таблице стилей браузера отсутствует значение этого свойства, оно наследуется естественным образом.)

3. *unset* – возвращает свойству его естественное значение, это означает что, если свойство наследуется естественным образом, оно действует как *inherit,* иначе оно действует как *initial*.

4. *revert* – откатывает один текущий уровень каскада, таким образом свойство принимает такое значение, которое было бы, если бы не было никаких стилей в текущем источнике стилей (авторских, пользовательских или браузерных).

И наконец последним технологическим китом из тройки является JavaScript.

JavaScript – это кросс-платформенный, объектно-ориентированный скриптовый язык, являющийся небольшим и легковесным. Внутри среды исполнения JavaScript может быть связан с объектами данной среды и предоставлять программный контроль над ними.

JavaScript включает стандартную библиотеку объектов, например, Array, Date и Math, а также базовый набор языковых элементов, например, операторы и управляющие конструкции. Ядро JavaScript может быть расширено для различных целей путём добавления в него новых объектов, например:

1. JavaScript на стороне клиента расширяет ядро языка, предоставляя объекты для контроля браузера и его Document Object Model (DOM). Например, клиентские расширения позволяют приложению размещать элементы в форме HTML и обрабатывать пользовательские события, такие как щелчок мыши, ввод данных в форму и навигация по страницам.
2. JavaScript на стороне сервера расширяет ядро языка, предоставляя объекты для запуска JavaScript на сервере. Например, расширение на стороне сервера позволяет приложению соединяться с базой данных, обеспечивать непрерывность информации между вызовами приложения или выполнять манипуляции над файлами на сервере.

На момент создания JavaScript имел другое имя – "LiveScript". Однако, язык Java был очень популярен в то время, и было решено, что позиционирование JavaScript как "младшего брата" Java будет полезно.

Со временем JavaScript стал полностью независимым языком со своей собственной спецификацией, называющейся ECMAScript, и сейчас не имеет никакого отношения к Java.

Современный JavaScript – это "безопасный" язык программирования. Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что изначально был создан для браузеров, не требующих этого.

Возможности JavaScript сильно зависят от окружения, в котором он работает. Например, Node.JS поддерживает функции чтения/записи произвольных файлов, выполнения сетевых запросов и т.д.

В браузере для JavaScript доступно всё, что связано с манипулированием веб-страницами, взаимодействием с пользователем и веб-сервером.

Например, в браузере JavaScript может:

1. Добавлять новый HTML-код на страницу, изменять существующее содержимое, модифицировать стили.
2. Реагировать на действия пользователя, щелчки мыши, перемещения указателя, нажатия клавиш.
3. Отправлять сетевые запросы на удалённые сервера, скачивать и загружать файлы (технологии AJAX и COMET).
4. Получать и устанавливать куки, задавать вопросы посетителю, показывать сообщения.
5. Запоминать данные на стороне клиента ("local storage").

Возможности JavaScript в браузере ограничены ради безопасности пользователя. Цель заключается в предотвращении доступа недобросовестной веб-страницы к личной информации или нанесения ущерба данным пользователя.

Примеры таких ограничений включают в себя:

1. JavaScript на веб-странице не может читать/записывать произвольные файлы на жёстком диске, копировать их или запускать программы. Он не имеет прямого доступа к системным функциям ОС.
2. Современные браузеры позволяют ему работать с файлами, но с ограниченным доступом, и предоставляют его, только если пользователь выполняет определённые действия, такие как "перетаскивание" файла в окно браузера или его выбор с помощью тега <*input*>.
3. Существуют способы взаимодействия с камерой/микрофоном и другими устройствами, но они требуют явного разрешения пользователя. Таким

образом, страница с поддержкой JavaScript не может незаметно включить веб-камеру, наблюдать за происходящим.

4. Различные окна/вкладки не знают друг о друге. Иногда одно окно, используя JavaScript, открывает другое окно. Но даже в этом случае JavaScript с одной страницы не имеет доступа к другой, если они пришли с разных сайтов (с другого домена, протокола или порта).

Это называется "Политика одинакового источника" (Same Origin Policy). Чтобы обойти это ограничение, обе страницы должны согласиться с этим и содержать JavaScript-код, который специальным образом обменивается данными.

Это ограничение необходимо, опять же, для безопасности пользователя. Например, страница *https://anysite.com*, которую открыл пользователь, не должна иметь доступ к другой вкладке браузера с URL *https://gmail.com* и воровать информацию оттуда.

5. JavaScript может легко взаимодействовать с сервером, с которого пришла текущая страница. Но его способность получать данные с других сайтов/доменов ограничена. Хотя это возможно в принципе, для чего требуется явное согласие (выраженное в заголовках HTTP) с удалённой стороной. Опять же, это ограничение безопасности.

В итоге каждая из технологических составляющих рассмотренных в данном подразделе невероятно важна для разработки современных веб-приложений. Они как единое целое взаимодействуют друг с другом, позволяя веб-страницам существовать и соответствовать требованиям современного пользователя. Каждый из технологических китов выполняя свои функции позволяет страницам внутри браузерной среды быть структурно-организованными используя HTML, визуально привлекательными благодаря CSS, а также быть функциональными используя JavaScript. Дальнейшее развитие этих трех фундаментальных технологий позволило веб-страницы стать полноценными приложениями.

1.3 Vue 3 как инструмент разработки SPA.

Рассмотренные в предыдущей главе фундаментальные веб-технологии являются на сегодняшний день лишь необходимой базой благодаря которой сама по себе разработка современных и сложных веб-приложений возможна, но не целесообразна в нынешних реалиях. Новые фреймворки и библиотеки вытеснили такую разработку в раздел ознакомления и обучения фронтенду, и они же позволили создавать веб-приложения в разы быстрее и, дешевле для рынка, это помогло вебу еще глубже проникнуть в жизни простых обывателей.

Из множества библиотек и фреймворков таких как React, Angular, Emberjs, Vue.js, Django, Laravel, Ruby, с помощью которых создаются современные веб-приложения рассмотрим Vue.js.

Vue (произносится /vjuː/, примерно как view) – прогрессивный фреймворк для создания пользовательских интерфейсов. В отличие от фреймворков-монолитов, Vue создавался пригодным для постепенного внедрения. Его ядро в первую очередь решает задачи уровня представления (view), упрощая интеграцию с другими библиотеками и существующими проектами. С другой стороны, Vue полностью подходит и для разработки сложных одностраничных приложений (SPA, Single-Page Applications), если использовать его в комбинации с современными инструментами и дополнительными библиотеками.

Vue 3 является логическим продолжением предыдущей версии фреймвока - Vuejs 2, первый релиз которого вышел еще в феврале 2014 года. Его создателем является Эван Ю (Evan You), который до этого работал в Google над AngularJS. С тех пор фреймфорк динамично развивается. В сентябре 2020 года вышла текущая версия фреймворка - Vue 3, особенностью которой стало то, что эта версия полностью написана на языке TypeScript вместо JavaScript, который использовался ранее.

Vue 3 имеет довольно небольшой размер и при этом обладает хорошей производительностью по сравнению с такими фреймворками как Angular или React, а также по сравнению с предыдущей версией - Vue.js 2.x.

Одним из ключевых моментов в работе Vue 3 является виртуальный DOM. Структура веб-страницы, как правило, описывается с помощью DOM (Document Object Model), которая представляет организацию элементов html на странице. Для взаимодействия с DOM (добавления, изменения, удаления html-элементов) применяется JavaScript. Но когда мы пытаемся манипулировать html-элементами с помощью JavaScript, то мы можем столкнуться со снижением производительности, особенно при изменении большого количества элементов. А операции над элементами могут занять некоторое время, что неизбежно скажется на пользовательском опыте. Однако если бы мы работали из кода js с объектами JavaScript, то операции производились бы быстрее.

Для этого Vue 3 использует виртуальный DOM. Виртуальный DOM представляет легковесную копию обычного DOM. Если приложению нужно узнать информацию о состоянии элементов, то происходит обращение к виртуальному DOM. Если данные, которые используются в приложении Vue 3, изменяются, то изменения вначале вносятся в виртуальный DOM. Потом Vue выбирает минимальный набор компонентов, для которых надо выполнить изменения на веб-странице, чтобы реальный DOM соответствовал виртуальному. Благодаря виртуальному DOM повышается производительность приложения.

Vue 3 поддерживается всеми основными современными браузерами. Фреймворк удобен также и потому, что его можно интегрировать в проект практически на любом этапе разработки. Для этого нужно всего лишь добавить следующий код в ваш HTML:

<script src="https://unpkg.com/vue@next"></script>

Такой способ установки можно использовать если необходимо добавить какой-либо небольшой функционал из Vue в уже работающий проект. Если же вам потребуется использовать его функционал в большем объеме же необходимо разработать весь проект на Vue, то тут лучше воспользоваться таким инструментом как VueCLI. Он позволит произвести разбивку всего проекта на

компоненты каждый из которых можно разместить в отдельном файле и даже сделать независимым от разрабатываемого проекта, чтобы использовать в своих будущих проектах. Также VueCLI удобен тем, что создаваемый проект четко структурирован, это помогает сделать разработку более наглядной и понятной не только для одного разработчика, но и делает более удобной работу в команде.

Установка VueCLI требует предустановку Node.js на компьютер. Для этого необходимо зайти на официальный сайт https://nodejs.org скачать сам Node.js и запустить установку.

Установить VueCLI глобально можно через ввод, следующий команды в консоль:

npm install -g @vue/cli

Также всегда можно воспользоваться установкой VueCLI локально в саму папку с проектом. Для этого в консоль нужно ввести следующую команду:

npx @vue/cli create YourFolderName

После ввода данной команды нужно пройти первоначальную настройку проекта, затем подождать пока локально установятся все необходимые библиотеки и можно начинать работу. После установки стартовые настройки проекта можно будет скорректировать изменив файл "package.json".

Структура создаваемого таким образом проекта может выглядеть так:

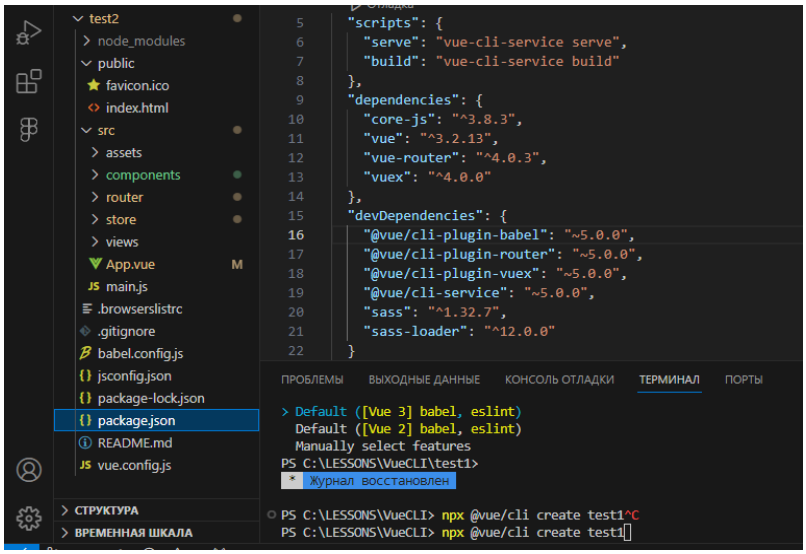


Рис.1 – Структура проекта

В корневой папке проекта создаются следующие директории:

1. node_modules – в этой папке лежат коды и исполняемые файлы установленных внешних (не наших) модулей, ее не стоит открывать и менять. Также ее не стоит коммитить в репозиторий (поэтому она по умолчанию была добавлена в файл .gitignore);
2. public – здесь лежат файлы, которые используются в проекте, но при сборке будут скопированы без изменений. Исключением является только файл index.html - во время компиляции в него добавятся тэги <script>, подключающие трансформированные файлы .js;
3. src – папка со исходным кодом проекта. Здесь будет лежать вся логика нашего приложения;
4. src/assets – здесь обычно лежат изображения и css файлы со стилями, которые используются в приложении;
5. src/components – папка, содержащая отдельные Vue компоненты в виде файлов с расширением vue;
6. src/main.js – входной файл проекта во Vue, мы будем его изменять, когда захотим что-нибудь подключить к нашему приложению, например – плагин;
7. src/App.vue – корневой компонент приложения, все остальные компоненты будут добавляться внутри него;
8. src/store – пака в которой располагаются файлы подключенной библиотеки Vuex, например, файл основного хранилища index.js и прочие файлы, позволяющие вынести отельные части основного файла как отдельные модули;
9. src/router – папка в которой находятся файлы подключенной библиотеки Router, которые отвечают за навигацию внутри SPA.

Рассмотрим компонентный подход более наглядно.

Однофайловые компоненты Vue (они же файлы *.vue, или называемые сокращённо SFC, от Single File Component) – специальный формат файлов,

позволяющий собрать в одном файле шаблон, логику и стилизацию компонента Vue. Пример однофайлового компонента представлен ниже:

```
<template>
    <p class="greeting">{{ greeting }}</p>
</template>
<script>
export default {
    data() {
        return {
            greeting: 'Привет всем!'
        }
    }
}
</script>
<style>
    greeting {
        color: red;
        font-weight: bold;
    }
</style>
```

Как можно увидеть, однофайловые компоненты Vue естественно развивают использование классического трио: HTML, CSS и JavaScript. Чаще всего *.vue файл состоит из трёх секций языковых блоков: <*template*>, <*script*> и <*style*>:

Секция <*script*> представляет собой обычный модуль JavaScript. Модуль в качестве экспорта по умолчанию должен возвращать объект с определением компонента Vue.

Секция <*template*> определяет шаблон компонента.

Секция <*style*> определяет CSS, связанный с компонентом.

Однофайловые компоненты Vue – формат, специфический для фреймворка, который нужно предварительно скомпилировать в обычный JavaScript и CSS с помощью *@vue/compiler-sfc*. Скомпилированный однофайловый компонент – это обычный ES-модуль JavaScript – а значит, при правильной настройке сборки, импортировать однофайловые компоненты можно как обычные модули:

```
<script>
    import MyComponent from './MyComponent.vue'
</script>
```

Секции <*style*> однофайловых компонентов во время разработки будут внедряться как нативные теги <*style*> для поддержки горячих обновлений. При сборке в production они могут быть извлечены и объединены в одном CSS-файле.

Однофайловые компоненты – определяющая особенность Vue как фреймворка и рекомендуемый подход для использования Vue в следующих сценариях:

1. Одностраничные приложения (Single-Page Applications, SPA)
2. Генерация статических сайтов (Static Site Generation, SSG)
3. Любые нетривиальные фронтенды, где этап сборки может быть оправдан для улучшения опыта разработки.

Другая не менее важная особенность Vue – это использование в HTML разметке компонента различных директив.

Директивы – это специальные атрибуты Vue которые именуются с префикса v- и добавляют особое реактивное поведение отрисованному DOM.

v-bind:attrName="value" – это директива которая позволяет связать значение какого-либо атрибута HTML-элемента (attrName) с изменяющимися данными (value). Т.к. это одна из наиболее часто используемых во Vue директив она получила краткую форму записи: название директивы "v-bind" можно не указывать, оставив лишь двоеточие перед названием связываемого атрибута.

v-on:eventName="handle('ok', $event)" – это директива позволяющая добавить обработчик события. Так же, как и v-bind является наиболее используемой и имеет сокращенный формат записи. Вместо "v-on" указывают символ @ перед названием события.

v-on – прикрепляет обработчик события к элементу. Тип события определяется аргументом (eventName). Выражение в кавычках может быть именем метода, инлайн-выражением или даже не указываться, при использовании модификаторов.

При использовании на обычном элементе отслеживает только нативные события DOM (opens new window). При использовании на компонентах отслеживает пользовательские события, которые были сгенерированы в нём.

При отслеживании нативных событий DOM в метод аргументом будет передаваться объект события. При указании инлайн-выражения, к объекту события можно получить доступ через специальное свойство $event: v-on:click="handle('ok', $event)".

v-text – это директива которая обновляет свойство HTML-элемента textContent. При необходимости обновления лишь части содержимого textContent лучше использовать интерполяцию с фигурными скобками – {{text}}.

```
<span v-text="msg"></span>
<!-- тоже самое что и -->
<span>{{ msg }}</span>
```

v-html – это директива которая обновляет свойство элемента innerHTML. Учтите, что содержимое будет вставляться как обычный HTML и не будет компилироваться или обрабатываться как шаблоны Vue.

v-show – это директива которая отображает элемент по условию, выполняя переключение у элемента CSS-свойства display в зависимости от истинности указанного выражения.

v-if – это директива которая отрисовывает элемент по условию, в зависимости от истинности указанного выражения. При переключении элемент и все содержащиеся в нём директивы / компоненты будут уничтожены и созданы заново. Для <*template*> будет отрисовываться его содержимое. Директива запускает анимации перехода при изменении состояния.

v-else-if, v-else – это директивы являющиеся логичным продолжением предыдущей директивы в списке, они отрисовывают альтернативную HTML-разметку. Таким образом используя v-if и v-else-if, v-else вместе в зависимости от истинности условия будет отрисован один из вариантов разметки.

v-for – это директива которая многократно отрисовывает элемент или блок шаблона на основании исходных данных. Значение директивы должно использовать специальный синтаксис alias in expression, чтобы объявить переменную для текущего элемента итерации:

```
<div v-for="item in items" :key="item.id">
        {{ item.text }}
</div>
```

По умолчанию v-for будет обновлять элементы "на месте", не перемещая их. Если необходимо переупорядочивать элементы при изменениях, то потребуется указывать специальный атрибут key.

v-model – это директива которая создаёт двухстороннюю привязку к элементу ввода формы или к компоненту. Используется вместе с компонентами *<input>*, *<select>* и *<textarea>*.

ГЛАВА 2

Особенности разработки интернет-магазина

2.1 Основные требования к приложению в процессе его разработки

Прежде чем приступить к практической реализации нашего приложения стоит для начала определиться с его назначением – что именно оно должно выполнять, архитектурой, структурными особенностями, функциональными возможностями, внешним видом, рассмотреть целый список требований характерных для современных веб-приложений.

Наверное, самым основным из требований, определяющих вектор разработки является именно назначение приложения. Исходя из него все вышеперечисленные требования будут варьироваться. Не трудно догадаться из темы Дипломной работы, что по назначению наше приложение будет являться интернет-магазином. Стоит уточнить сразу, что в магазине будут продаваться потребительские товары различных категорий: одежда, обувь, украшения, декор, техника и прочие товары, т.к. существует масса различных интернет-магазинов торгующих, например, только одной категорией товаров или теми в которых представлены оказываемые услуги. Как по существу, так и по масштабу наш магазин не будет сильно отличаться от тех, в которых представлена одна категория товаров, т.к. и там, и здесь все равно есть разделение на более мелкие категории и подкатегории, фильтрация и поиск товаров, к тому же типов реализуемой продукции будет не много.

Архитектура магазина также, как и его назначение указана в теме Дипломной работы – а именно Single Page Application. Учитывая все достоинства и недостатки SPA, она очень неплохо подходит для создания небольших приложений подобных нашему. К тому же от интернет-магазина требуется быстрое перемещение между страницами внутри самого приложения, тогда как скоростью первой загрузка можно и пожертвовать.

Структурно приложение будет состоять из набора Vue-компонентов, взаимодействующих между собой через Vue-router и Vuex, также будет использован LocalStorage.

Среди основных компонентов магазина будут следующие:

1. Header сайта, который будет общим для всех страниц;
2. Главная страница сайта, на которой будет отображен основной контент приложения и некоторые категории товаров;
3. Страница каталога, где будет можно будет как посмотреть все доступные товары магазина, так и осуществить фильтрацию по определенному признаку;
4. Три информационные страницы сайта, без сложного функционала, из которых можно будет узнать всю необходимую для пользователя информацию: О магазине, Доставка и Контакты;
5. Страница входа-регистрации пользователя;
6. Страница избранных товаров;
7. Корзина товаров;
8. Личный кабинет пользователя, в котором можно будет посмотреть, как текущие заказы, так и товары в избранном и корзине, а также изменить регистрационные данные;
9. Footer сайта, также как и header будет общим для всех страниц.

Серверная часть приложения будет представлена веб-сервисом DummyJSON. Это псевдосервер, который способен получая запросы как настоящий сервер предоставлять определенные данные в JSON-формате. Отсюда мы будем получать данные для формирования карточек товаров. Здесь их будет доступно ровно сто штук, этого достаточно для разработки основного функционала приложения. Также сервер способен присылать наборы карточек по категориям, чем мы и будем пользоваться при сортировке товаров.

Функционально Интернет-магазин должен предоставлять следующие возможность пользователю:

1. Осуществлять вход-регистрацию, клиента на сайте;
2. Предоставлять все товары по запросу;
3. Предоставлять возможность фильтрации товаров и сортировки;
4. Отображать подробности о товаре по клике;
5. Осуществлять поиск конкретного товара по названию;
6. Добавление товаров в избранное;
7. Добавление товаров в корзину;
8. Изменение личных данных;
9. Возможность просмотра списка текущих заказов;

По внешнему виду сайта можно сказать следующее: во-первых, цветовая гамма не должна быть "вырвиглазной" т.е. сайт должен быть сочетаем по цветам и иметь собственный стиль; во-вторых, следует избегать информационного перегруженности сайта, например, стоит избегать избыточного информирования в карточке товара т.к. детальная информация о нем будет отображена на странице самого товара. Более подробно дизайн будет рассмотрен в соответствующем пункте Дипломной работы.

Еще одним стандартным, но не менее важным набором требований к современному приложению являются кросбраузерность и адаптив. В обеспечение кросбраузерности приложения частично помогает фреймворк Vue 3, т.к. имеет широкую поддержку среди браузеров и при сборке добавляет в проект необходимые полифилы и префиксы под диапазон браузеров указанных в поле browserslist в package.json файле. Для реализации адаптивных возможностей приложения можно воспользоваться медиазапросами.

В итоге определившись с перечнем требований к веб-приложению можно приступать непосредственно к разработке приложения.

2.2 Дизайн и структура страниц Интернет-магазина

Рассмотрим подробнее дизайн и структуру Интернет магазина.

Как правило Frontend-разработчик создает приложение по готовому дизайн-макету каждой из страниц веб-приложения. Обычно макет подготавливается UX/UI-дизайнером, и для того, чтобы разработать внешний вид всего приложения придется на время им стать.

UX-дизайнер – специалист, задача которого сделать продукт удобным, понятным и логичным для всех пользователей. От его работы зависит, сможет ли клиент быстро и удобно получить желаемую услугу.

UI – user interface – это пользовательский интерфейс: наполнение сайта, систематизация элементов, выбор цветов, построение визуальной композиции, оформление кнопок, колонок и других графических элементов.

Для начала определимся с цветовой гаммой приложения. В качестве основных цветов всего проекта я выбрал комбинацию оттенков зеленого, черного и белого цветов.

Например, элементы Header и Footer будут окрашены в черный и серые цвета. Цвет шрифта и иконок будет инверсированы, если фон черный, то текст серый и наоборот, при неведении цвет будет становится зеленым.

Для контентной части страницы основными цветами станут светло-зеленый, белый и серый, шрифт будет черного цвета.

Основными шрифтами для приложения решено использовать "oxygen-regular" для простого текста, "cabin-700", "cabin-500" для заголовков.

Приступим непосредственно к созданию дизайна на примере главной страницы нашего приложения. Стоит начать именно с нее, т.к. все остальные страницы будут в той или иной степени заимствовать ее стиль.

Структуру главной страницы сделаем так как показано на рисунке 2, на следующей странице. Помимо элементов Header и Footer на главной будут Main-секция с динамически меняющимися задним фоном, секция с ссылками на стр. каталога с карточками, отфильтрованными по названиям категорий, также будут 3 секции с карточками: новинки, бестселлеры и рекомендуем.

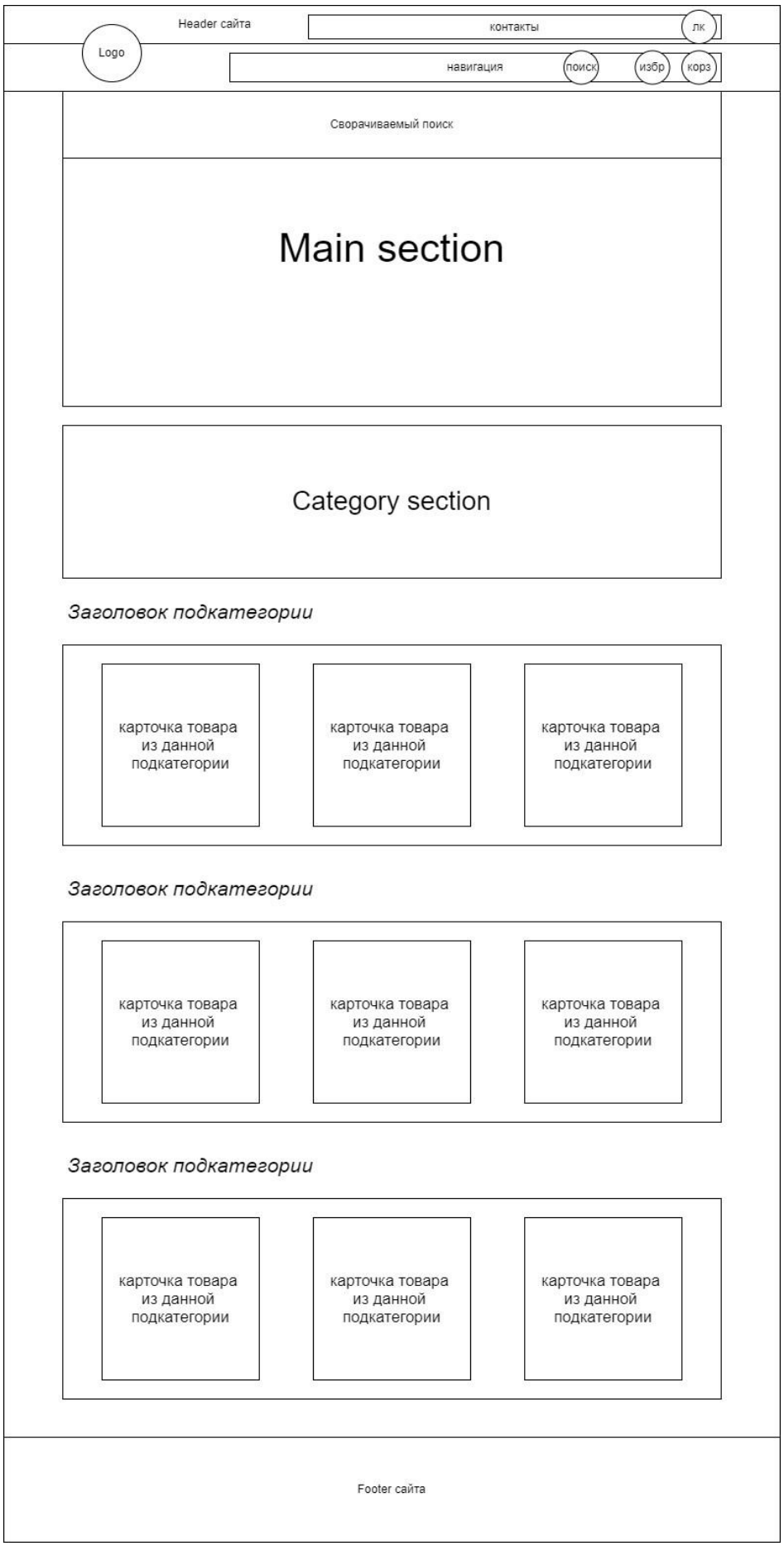


Рис. 2 – Эскиз главной страницы сайта

Эскизы остальных страниц сайта представлены в Приложениях 1-11 к дипломной работе. Нет необходимости перечислять абсолютно все дизайнерские задумки т.к. многие из них вы и так сможете увидеть в самом веб-приложении, но некоторые из них стоят того чтобы быть упомянутыми в работе.

Интернет-магазин – это в первую бренд, который должен иметь в своей основе какую-нибудь историю или концепцию. Суть концепции нашего бренда отражена в его название "AllWaysOnline". Само название как бы говорит потребителю, что магазин всегда доступен, он всегда онлайн. Также в название кроется второй скрытый смысл, запрятанный в его словах "Ways" – в переводе с англ. пути и "Online". Как все дороги вели в Рим в далекие времена античности, так и в наше время все дороги ведут в Интернет в "Online".

Логотип для сайта был разработан в веб-приложении Figma и размещен в шапке нашего сайта. Рисунок логотипа Интернет-магазина вы можете видеть на следующей иллюстрации:

Рис. 3 – Логотип интернет-магазина

Одним из самых главных элементов любого интернет-магазина является карточка товара. Окончательный дизайн карточки представлен на рисунке 3.

Рис.4 – Карточка товара

На карточке товара представлена только самая необходимая информация о товаре и его статусе для покупателя. Основную часть карточки занимает картинка с изображением товара поверх которого расположен стилистический элемент, содержащий информацию о категории к которой относится товар. Ниже в информационном блоке содержится основная информация для покупателя: название товара, его бренд, цена и цена со скидкой. В самом низу карточки расположены ее функциональные элементы: кнопка подробнее – ссылка на страницу товара с подробной информацией, иконки избранное и корзина которые отображают статус товара для покупателя, если товар добавлен в избранное и/или в корзину иконка станет красной.

В итоге разработка дизайна элементов интернет-магазина являются самым первым шагом на пути создания приложения. Именно дизайн задает структуру страниц и компонентов всего приложения.

2.3 Структура Vue-приложения Интернет-магазина. Связь компонентов между собой через Vue-router, Vuex.

Рассмотрев основные требования к веб-приложению и его дизайнерские особенности можно переходить непосредственно к разработке. Создание нового приложения с использованием инструментов VueCLI всегда начинается с базовой структуры, которая кратко была уже рассмотрена ранее в п. 1.3. Что ж, давайте рассмотрим ее более детально.

В директории "src" расположены основные файлы и папки проекта.

App.vue – корневой компонент, который по сути и является основой всего приложения. По структуре он состоит всего лишь из трех компонентов: шапки сайта, компонента страницы, который меняется с помощью роутера и подвала сайта, но при этом связан со всеми компонентами страниц. В момент своего создания корневой компонент инициализирует все основные данные приложения и отображает стартовую страницу.

Файл main.js – основной скрипт приложения. Он содержит в себе всего лишь 7 строк кода:

1. *import { createApp } from "vue";*
2. *import App from "./App.vue";*
3. *import "@/assets/styles/styles.scss";*
4. *import router from "@/router/router";*
5. *import store from "./store";*
6.
7. *createApp(App).use(store).use(router).mount("#app");*

Первые 5 строк импортируют в приложение основные модули общие стили всего приложения, а также библиотеки Vue-router, Vuex-store. В седьмой строке создается сам экземпляр Vue и монтируется в корневой элемент *#app* вместе с объектами *store* и *router*.

Папка "assets" также расположена в директории "src". Она включает в себя все основные ресурсы приложения в частности папку со всеми картинками

приложения "img" и папку "styles" в которой расположены все общие стили для каждого из компонентов приложения.

Папка "router" содержит в себе всего один файл "router.js". В этом файле создаются сам *router* который, затем импортируется в основной скрипт *main.js* и массив объектов-маршрутов *routes*,

Каждый объект внутри массива имеет минимум 2 свойства путь в поисковой строке браузера и название компонента который будет отрисован в корневом компоненте на месте элемента *<router-view> </router-view>*. Соответственно, для того, чтобы все это работало в файл, импортируются все эти компоненты а также 2 основные функции *createRouter()* и *createWebHistory()*.

Код создания роутера выглядит следующим образом:

```
const router = createRouter({
        history: createWebHistory(),
        routes,
        scrollBehavior(to, from, savePosition) {
                return {
                        top: 0,
                };
        },
});
// Экспорт роутера
export default router;
```

Здесь функция *createRouter()* принимает объект в котором создается история перехода по ссылкам *createWebHistory()*, содержится массив маршрутов *routes*, а также задается поведение при прокрутке страницы с помощью метода *scrollBehavior()*. В нашем случае *scrollBehavior()* установит начальную координату прокрутки равную нулю при переходе на новую страницу.

Папка "store" является общим хранилищем данных необходимых для функционирования приложения. Она содержит в себе два файла "index.js" и "localStore.js".

Index.js является центром всего приложения, местом где находится хранилище Vuex, а "localStore.js" – это модуль, который содержит весь функционал взаимодействия приложения с LocalStorage браузера.

"Хранилище" – это, по сути, контейнер, в котором хранится состояние приложения. Оно содержит:

1. Основные данные приложения, называемые состоянием (state) – это просто объект с данными;
2. Геттеры(getters) – вычисляемые из данных состояния свойства;
3. Мутации (mutations) – методы, которые изменяют состояние,
4. Экшены (actions) – асинхронные методы, изменяющие состояние;
5. Модули (modules) – небольшие хранилища, вынесенные в отдельные файлы.

Все взаимодействие компонентов между собой сведено к минимуму, по сути они связаны между собой с помощью store, и берут все необходимые данные из него. При этом Vuex моментально реагирует на изменения состояния. Когда компоненты Vue извлекают какие-либо данные из него, то эти данные будут оперативно и эффективно обновлять и состояние самого хранилища.

Первая особенность Vuex в том, что напрямую изменить состояние приложения нельзя. Единственный способ его изменить – явно зафиксировать мутации. Это гарантирует, что каждое изменение состояния будет отслеживаемым.

Вторая особенность Vuex в том, что для асинхронных методов следует использовать экшены вместо мутации.

Папка "pages" содержит в себе основные страницы приложения – крупные компоненты по которым Vue- router осуществляет переходы.

MainPage – компонент, главной страницы сайта, он имеет начальный путь "path: \" в роутере и является стартовой страницей. Она напрямую связана со страницей каталога, через секцию категорий и три секции товаров: новинки, бестселлеры и рекомендуем. При клике на категорию товара или на кнопку "подробнее" одной из трех вышеперечисленных секций будет производится переход на страницу каталога, где будут отображаться карточки товаров в соответствии с выбранной категорией или секцией, в которой товары отсортированы по одному из трех признаков.

AllProductPage, LikedPage, SearchResultPage – три страницы схожие по дизайну и функционалу. Это как раз-таки та самая страница каталога, связанная с главной страницей, а также страница избранных товаров и страница результатов поиска. Их главная задача отображать карточки товаров. Разница лишь в том, что первая отображает все товары динамически подгружая их с сервера, а также может отображать товары, отфильтрованные по категориям. В то время как LikedPage и SearchResultPage лишены возможности фильтрации и отображают лишь товары одного типа соответственно это избранные товары и результаты поиска товаров по названию. Через компонент карточки товара, который будет рассмотрен далее, каждая из страниц ссылается на отдельный товар, а путь к нему задается динамически с помощью id товара.

ProductPage – компонент страницы позволяющий отобразить более подробную информацию о товаре. По сути каждая карточка товара ссылается на этот компонент, а он в свою очередь позволяет отобразить информацию о том товаре id которого указан в поисковой строке браузера. Со страницы товара можно также добавить или убрать продукт из корзины.

DeliveryPage, ContactsPage, AboutUsPage – три информационные страницы которые должны содержать любой контент предоставляемый заказчиком сайта, а именно: информацию о доставке, контактную информацию и какой-либо контент про магазин. В будущих подразделах эти страницы рассматриваться не будут, т.к. они не содержат сложного функционала и имеют простую структуру.

С эскизами страниц можно будет ознакомиться в Приложении к дипломной работе № 2 – 4.

AuthPegistrationPage – компонент страницы входа или регистрации. Эта страница взаимодействуют с LocalStorage браузера и узнает был пользователь зарегистрирован на сайте или нет, в зависимости от чего и отображает формы входа или регистрации. Она также очень тесно связана с личным кабинетом пользователя. Если во время входа в личный кабинет пользователь поставит галочку на чекбокс "Запомнить меня", то при каждой следующей загрузке приложения вместо страниц входа/регистрации сразу будет отображаться личный кабинет.

AccountPage – это тот самый компонент который является личным кабинетом пользователя. Он выполняет 4 основные функции:

1. Отображает все текущие заказы клиента, все товары в каждом из них с ценной, названием, количеством и общую стоимостью каждого заказа;
2. Позволяет просматривать и редактировать личные данные, указанные в момент регистрации;
3. Отображает все избранные товары;
4. Отображает все товары, находящиеся в корзине.

BasketBage – это корзина товаров клиента. Она не только отображает все товары, добавленные в корзину, но также позволяет отредактировать количество товаров в заказе, и показать их итоговую стоимость. При клике по кнопке заказать, в корзине формируется текущий заказ, который будет доступен в личном кабинете, после чего все товары из самой корзины будет удалены.

ErrorPage – это страница которая будет отображаться в случае ошибки или перехода на несуществующую страницу.

И наконец рассмотрев основные страницы Интернет-магазина перейдем к менее масштабным, но не менее важным компонентам приложения.

Папка "components" содержит переиспользуемые на основных страницах компоненты, они как правило меньше чем компоненты страниц. Вот основные из них:

HeaderComp – шапка сайта, он находится в корневом компоненте и загружается один раз вместе с ним. Этот компонент не только отвечает за навигацию по основным разделам сайта, в нем также расположен поисковой элемент приложения, с помощью которого пользователь может находить товары по названию и отображать результаты поиска на странице SearchResultPage.

FooterComp – подвал сайта, он также находится в корневом компоненте и загружается вместе с шапкой сайта всего один раз. Расположение этих двух компонентов в App.vue позволяет избежать многократного дублирования, и немного увеличить скорость загрузки страниц внутри приложения т.к. на каждой странице сайта шапка и подвал абсолютно одинаковые.

CartProduct – это компонент товара, внутри корзины товаров. С его помощью можно менять количество данной позиции товара, либо удалить товар из корзины целиком.

SecondCard – компонент карточки товара. Это, пожалуй, самый важный, сложный по своему функционалу и наиболее часто используемый компонент приложения. Он не только отображает основную информацию по конкретному товару, но также обязан взаимодействовать с локальными данными браузерами, где хранится список избранных товаров и товаров в корзине. Это необходимо потому, что везде где бы не отображалась карточка товара у пользователя должна быть возможность по клику на иконки сердечка или корзинки добавить, или убрать товар из списка избранных или корзины. Также в момент своего создания компонент должен понимать добавлен ли отображаемый товар в один из таких списков, и отобразить это визуально.

SwipperAutoplay – это компонент созданный на основе всемирно-известной библиотеки Swipper.js. Он всего лишь предоставляет возможность

пользователю переключать картинки в блоке Main на главной странице, либо переключает ее раз в 15 сек. сам автоматически.

В итоге все компоненты приложения связаны между собой, как и напрямую, когда какой-либо из-них содержит в себе несколько других так и косвенно, когда компоненты меняют данные друг друга без прямого взаимодействия. Наладить связь между компонентами позволяют две основные библиотеки Vue-router и Vuex. Также базовая структура файлов проекта позволяет сделать разработку на Vue более понятной и четко структурированной.

ГЛАВА 3.

Реализация основного функционала Интернет-магазина

3.1 Корневой компонент App.vue и методы взаимодействия приложения с сервером.

Рассмотрим подробнее главный компонент всего приложения. Его шаблон содержит следующий код:

```
<template>
  <HeaderComp />
  <router-view v-slot="{ Component, route }">
        <component :is="Component" :key="route.path" />
  </router-view>
  <FooterComp />
</template>
```

Как уже говорилось ранее App.vue содержит всего три компонента. Шапку сайта, компонент страницы и подвал сайта. Также из предыдущего раздела известно, что основные переменные, в которых хранятся данные всего приложения создаются внутри Vuex-store, но изначально эти переменные заполняются либо пустыми значениями, либо значениями по умолчанию. Необходимыми данными Vuex-store заполняют уже сами компоненты.

На этапе создания приложения App.vue вызывает некоторые свои методы, позволяющие заполнить хранилище всеми необходимыми для загрузки стартовой и некоторых других страниц данными. Для этого используются хук жизненного цикла компонента "created". Код для корневого компонента будет выглядеть следующим образом:

```
created() {
    this.getSortedProducts({
        commitName: "addCardsInNewProducts",
        sortParametr: "id",
        startIndex: 0,
        endIndex: 18,
    });
    this.getSortedProducts({
```

```
            commitName: "addCardsInMostRatedProducts",
            sortParametr: "rating",
            startIndex: 0,
            endIndex: 18,
        });

        this.getSortedProducts({
            commitName: "addCardsInRecomendProducts",
            sortParametr: "discountPercentage",
            startIndex: 0,
            endIndex: 18,
        });
        this.getCards({ limit: 18, skip: 0 });
        this.readLocalUserData();
        this.getLikedProducts();
        this.getBasketProducts();
        this.getUserOrders();
    },
```

Здесь три раза вызывается метод *getSortedProducts()* с разными аргументами: название мутации, параметр сортировки, начальный и конечный индексы. Этот метод является асинхронным и импортируется из хранилища index.js. Суть заключается в том, на главной странице сайта располагается три секции с товарами, отсортированными по одному из трех показателей доступному нам из карточки товаров получаемых с сервера. В данном случае карточки отсортированы:

По новизне, здесь в качестве параметра используется id товара, чем товар позже появился на сайте, тем id будет выше;

По рейтингу товара, чем он выше, значит тем сильнее доволен покупатель своей покупкой;

По проценту скидки, чем выше процент, тем чаще такой товар будет рекомендоваться к покупке.

Разумеется, что такая сортировка возможна т.к. внутри метода *getSortedProducts()* мы направляем запрос, по которому от JSON сервера получаем массив объектов из id и значения параметра, по которому должна производится сортировка. Далее этот массив сортируется встроенным методам

массивов arr.sort(), а затем циклически пройдя по отсортированному массиву, запрашивается карточка товара по id. Сам метод выглядит так:

```
async getSortedProducts(
    { state, commit },
    { commitName, sortParametr, startIndex, endIndex }
  ) {
    try {
      const response = await fetch(
        `${state.serverUrl}/products?limit=${0}&select=${sortParametr}`
      );
      const result = await response.json();
      const sorted = result.products.sort(
        (a, b) => b[sortParametr] - a[sortParametr]
      );
      for (let i = startIndex; i < endIndex; i++) {
        try {
          const response = await fetch(
            `https://dummyjson.com/products/${sorted[i].id}`
          );
          const result = await response.json();
          commit(commitName, result);
        } catch (e) {
          console.log(e.message);
        }
      }
    } catch (e) {
      console.log(e.message);
    }
  },
```

Далее каждая запрошенная по id карточка товара добавляется в соответствующий для нее массив внутри index.js с помощью мутации название которой *commitName* передается в метод одним из аргументов. Такой метод сортировки может показаться громоздким и неоптимальным, но к сожалению, он единственно возможный, в данном случае т.к. функционал псевдосервера сильно ограничен, и по-хорошему товары уже должны приходить с сервера отсортированные по определенному признаку.

Следующим в хуке *created()* вызывается метод *getCards()* он является асинхронным и загружает указанное количество товаров. Он также импортируется вместе с остальными action в App.vue.

```
async getCards({ state, commit }, { limit, skip }) {
    try {
      const response = await fetch(
        `${state.serverUrl}/products?limit=${limit}&skip=${skip}`
      );
      const result = await response.json();
      commit("addCardsInArticles", result.products);
    } catch (e) {
      console.log(e.message);
    }
  },
```

Этот метод формирует запрос на сервер с помощью шаблонной строки, здесь *limit* – это количество получаемых с сервера товаров, а *skip* – это позиция, номер *id* с которой будет начинаться загрузка. По сути тут просто используются встроенные возможности JSON-сервера, позволяющие загрузить как все товары, так и часть из них. Параметр *skip* нам полезен тем, что позволяет реализовать динамическую подгрузку товаров на странице каталога, вместо того, чтобы получать сразу все. Мы просто передаем другие *{ limit, skip }* в метод и получаем следующие N-товаров.

Следующий метод *readLocalUserData()* отвечает за взаимодействие приложения с LocalStorage, его код выглядит следующим образом:

```
readLocalUserData(state) {
    const userData = JSON.parse(localStorage.getItem("userData"));
    userData && this.commit("user/changeUserData", userData);
    userData || this.commit("user/writeLocalUserData");
  },
```

Здесь сначала *localStorage.getItem* считывает данные из хранилища браузера, затем в виде объекта они сохраняются в объект *userData.* Далее если данные были считаны, они будут записаны в *user/changeUserData* в модуль *Vuex-store* под названием *localStore.js.* Если же данных в *localStorage* не оказалось, то в хранилище будут записаны дефолтные данные пользователя.

Следующие два метода выполняют схожие функции; они по сути получают массив id избранных товаров или товаров в корзине, загруженных из локального хранилища предыдущим методом, и направляют запрос на

получение товара по id с помощью асинхронного метода *getSingleProduct()* импортируемого из *index.js*. Код всех трех методов приведен ниже:

```
getLikedProducts() {
  this.delCardsInLiked();
  for (const id of this.likedProducts) {
   this.getSingleProduct({ id: id, commitName: "addCardInLiked" });
  }
},
getBasketProducts() {
  this.delCardsInBasket();
  for (const item of this.baskedProducts) {
   this.getSingleProduct({ id: item.id, commitName: "addCardInBasket" });
  }
}
// action-метод из файла index.js
async getSingleProduct({ state, commit }, { id, commitName }) {
   try {
     const response = await fetch(`${state.serverUrl}/products/${id}`);
     const result = await response.json();
     commit(commitName, result);
   } catch (e) {
     console.log(e.message);
   }
},
```

И наконец метод *getUserOrders()* обращается к массиву заказов покупателя и формирует список текущих заказов клиента. Код метода приведен ниже:

```
getUserOrders() {
     this.clearOrders();
     for (const order of this.currentOrders) {
       this.getUserOrderProduct(order);
     }
   },
async getUserOrderProduct({ id, status, sum, products }) {
     const tempProducts = [];
     for (const item of products) {
       try {
         const response = await fetch(`${this.url}/products/${item.id}`);
         const result = await response.json();
         result.quantity = item.value;
         tempProducts.push(result);
       } catch (error) {
```

```
        console.log(error.message);
      }
    }
    this.addInOrders({ id, status, sum, products: tempProducts });
  },
```

Сформированные таким образом заказы будут отображаться в личном кабинете пользователя, при загрузке страницы.

3.2 Компонент главной страницы приложения

Структура элементов главной страницы была уже рассмотрена в п.п. 2.2 и п.п. 2.4 поэтому здесь рассматривать ее повторно, нет никакого смысла, главное вспомнить, что она содержит основную секцию, за которую отвечает компонент Swipper.vue, секцию категорий товаров, состоящую из картинок-ссылок, и три секции сортированных товаров, принципы сортировки были рассмотрены в предыдущем подразделе. Основной код компонента необходимы для понимания принципов его работы предоставлен ниже:

```
export default {
  name: "MainPage",
  components: {
    SecondCard,
    GreenButton,
    SwipperAutoplay,
  },
  methods: {
   ...mapMutations({
        addCardsInArticles: "addCardsInArticles",
        delCardsInArticles: "delCardsInArticles",
    }),
   ...mapActions({
        getProductsForCategory: "getProductsForCategory",
    }),
    clickForCategory(category) {
      this.delCardsInArticles();
      for (const sub of category.subkategory) {
        this.getProductsForCategory(sub.en);
      }
      this.$router.push(`/products/${category.name.ru.toLowerCase()}`);
    },
    clickForSortedProducts(sortedProducts, title) {
      this.delCardsInArticles();
      this.addCardsInArticles(sortedProducts);
      this.$router.push(`/products/${title.toLowerCase()}`);
    },
  },
  computed: {
   ...mapState({
        categoryGrid: (state) => state.categoryGrid,
        newProducts: (state) => state.newProducts,
```

```
      mostRatedProducts: (state) => state.mostRatedProducts,
      recomendProducts: (state) => state.recomendProducts,
    }),
  },
};
```

Все основные данные для главной страницы уже были получены в корневом компоненте, и поэтому основная задача страницы просто импортировать эти данные из Vuex-store. Они добавлены в качестве вычисляемых свойств компонента внутри *computed.*

По коду видно, что компонент импортирует две мутации *addCardsInArticles()* и *delCardsInArticles()*, и один *action*-метод *getProductsForCategory()*, которые будут использованы в его собственных методах *clickForCategory()* и *clickForSortedProducts().*

Метод *clickForCategory()* работает следующим образом: по клику на картинку-ссылку категории сначала вызывается мутация *delCardsInArticles()* она отчищает массив карточек который страница каталога импортирует как вычисляемое свойство из Vuex-store. Затем внутри цикла for вызывается *action*-метод *getProductsForCategory()* который запрашивает с псевдосервера все карточки товаров подкатегории. Цикл for необходим потому, что JSON-сервер предоставляет нам около 19 подкатегорий, которые искусственно объединены на главной странице в 6 крупных категорий. Таким образом, чтобы получить все карточки "большой" категории необходимо сделать несколько запросов по каждой из подкатегорий.

Код *action*-метода *getProductsForCategory()* представлен ниже:

```
async getProductsForCategory({ state, commit }, categoryName) {
  try {
    const response = await fetch(
      `${state.serverUrl}/products/category/${categoryName}`
    );
    const result = await response.json();
    commit("addCardsInArticles", result.products);
  } catch (e) {
    console.log(e.message);
  }
},
```

Этот метод направляет запрос на сервер, передавая изменяемый URL внутри шаблонной строки. Это позволяет, используя всего один метод делать разные запросы и получать набор карточек только той категории, которая передана в качестве аргумента в *async*-функцию.

После того как цикл *for* полностью отрабатывает мы передаем во Vue-router команду, которая осуществляет переход на страницу каталога. Также стоит отметить, что адрес страницы каталога может принимать динамический параметр, что позволяет менять адрес в поисковой строке в зависимости от названия категории. Главный заголовок этой страницы также динамически изменяется.

Метод *clickForSortedProducts()* делает нечто похожее, что и *clickForCategory()* но в целом выполняет меньше работы. Он также отчищает массив товаров с помощью *delCardsInArticles()*, и заполняет его элементами одного из трех массивов отсортированных товаров. Затем также через Vue-router осуществляется переход на страницу каталога.

3.3 Компоненты страниц каталога, избранных товаров, поиска товаров.

Компоненты страниц каталога уже были кратко рассмотрены в п.п. 2.4, для того чтобы лучше разобраться в работе приложения давайте же проведем более подробный обзор этих трех схожих по между собой компонентов.

Для начала предыдущих подразделов известно, что исходные данные для страниц каталога и избранных товаров генерируются в корневом компоненте. Это позволяет просто обратится к *mapState* получить оттуда массив товаров *articles* и сразу отрисовать все карточки товаров на странице. Для каталога это выглядит следующим образом:

```
computed: {
   ...mapState({
     articles: (state) => state.articles,
     categories: (state) => state.categoryGrid,
     filters: (state) => state.filters,
    }),
  },
```

Также из *mapState* мы получаем список категорий *categories*, для фильтрации и изначально пустой массив *filters*. Список категорий и подкатегорий сразу отрисовывается на странице в элементе *Details* (Рисунок 5) который используется для фильтрации товаров.

Рис. 5 – Фильтрация товаров на странице. Нет выбранных фильтров.

Для взаимодействия изменения данных хранилища страница каталога импортирует мутации и экшены из файла *index.js*. Код представлен ниже:

```
...mapMutations({
     changeFilters: "changeFilters",
     delCardsInArticles: "delCardsInArticles",
    }),
```

```
...mapActions({
    getCards: "getCards",
    getProductsForCategory: "getProductsForCategory",
    getSortedProducts: "getSortedProducts",
  }),
```

Фильтрация товаров на странице работает следующим образом по клику на элемент *Details* – выбрать фильтр, открывается список категорий, которые также содержат в себе чекбоксы подкатегорий. Все чекбоксы связаны директивой *v-model* с массивом *checkedNames* в котором реактивно отображаются названия всех категорий, на которых пользователь поставил галочку. Фильтрация изображена на рисунке 6.

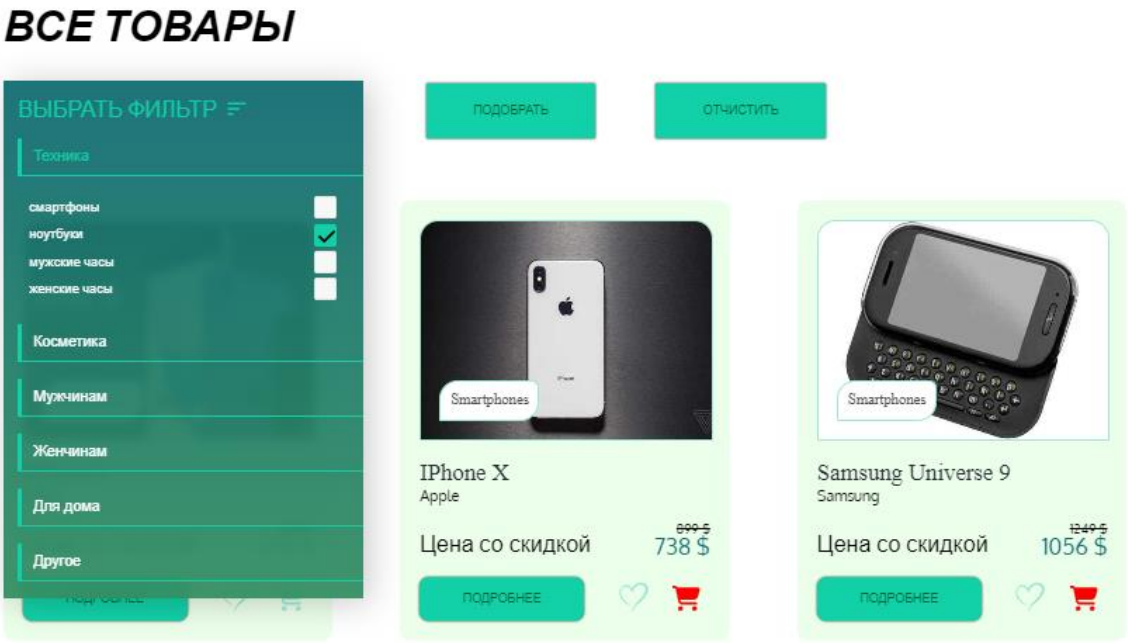


Рис. 6 – Меню фильтрации открыто

После того как все фильтры выбраны нужно всего лишь нажать кнопку "Подобрать" и тут в дело вступают методы компонента страницы *addFilters()* и *filtered()* код которых представлен ниже:

```
addFilters(checkedNames) {
    this.changeFilters(checkedNames);
    this.filtered();
  },
```

```
filtered() {
  this.delCardsInArticles();
  for (const filter of this.filters) {
    this.getProductsForCategory(filter);
  }
},
```

Метод *addFilters()* через мутацию *changeFilters()* записывает массив выбранных фильтров в массив *filters* в хранилище данных и вызывает метод *filtered()*. Далее внутри функции вызывается мутация *delCardsInArticles()* которая отчищает массив карточек *articles* и в цикле загружает карточки всех выбранных пользователем категорий с помощью *action*-метода *getProductsForCategory()*, который был рассмотрен в п.п. 3.2.

Кнопка "Отчистить" вызывает метод *clearFilters()* который отчищает массив *checkedName*s, затем через мутацию *delCardsInArticles()* отчищает массив карточек *articles* и наконец вызывает *action*-метод *getCards()* который подгружает базовый список товаров с сервера. Код метода представлен ниже:

```
clearFilters() {
    this.checkedNames.splice(0, this.checkedNames.length);
    this.delCardsInArticles();
    this.getCards({ limit: 18, skip: 0 });
  },
```

Как уже говорилось ранее на странице каталога реализована динамическая подгрузка товаров. Как видно из аргументов функции *getCards({ limit: 18, skip: 0 })* первоначально загружается 18 карточек начиная с нулевой. Для того, чтобы загрузить следующие 18 товаров пользователь должен нажать на кнопку "Загрузить" внизу страницы, вызвав тем самым метод *getNextCards()*. Код метода представлен ниже:

```
getNextCards() {
    const condition = ["новинки", "бестселлеры", "рекомендуем"]
      .indexOf( this.$route.params.category);
    if (this.$route.params.category === "все товары") {
      this.getCards({ limit: this.limit, skip: (this.skip += 18) });
    } else if (condition >= 0) {
      this.getSortedProducts({
        commitName: "addCardInArticles",
```

```
      sortParametr: ["id", "rating", "discountPercentage"][condition],
      startIndex: (this.startIndex += 18),
      endIndex: (this.endIndex += 18),
    });
  }
},
```

Метод реализует два основных типа подгрузки товаров. Во-первых, он позволяет загружать в порядке сортировки карточки из секций "новинки", "бестселлеры", "рекомендуем". Во-вторых, он реализует дефолтную для псевдосервера подгрузку товаров, реализованную через *action*-метод *getCards()*. Код метода был рассмотрен в п.п 3.1.

Суть метода *getNextCards()* заключается в том, что в блоке *if-else* проверяется следующее условие: *this.$route.params.category === "все товары"*, которое буквально проверяет что если динамический параметр пути страницы равен значению "все товары", то реализуется дефолтная подгрузка предоставляемая самим JSON-сервером. Если же условие ложно, то реализуется кастомная подгрузка с помощью метода *getSortedProducts()*, который был уже рассмотрен ранее в п.п.3.1.

Страница избранных товаров, по своей структуре очень похожа на каталог, с той разницей, что на ней нет фильтрации и динамической подзрузки товаров. Все данные для компонента формируются в корневом компоненте сама же страница просто отображает список товаров. Вся логика компонента выглядит следующим образом:

```
export default {
  name: "LikedPage",
  components: {
    SecondCard,
  },
  computed: {
    ...mapState({
      liked: (state) => state.liked,
    }),
  },
};
```

Как уже говорилось ранее в п.п. 2.3. пользователь может добавлять и удалять товары из избранных и корзины через саму карточку товара, а значит и добавлять дополнительные методу на страницу избранные не имеет смысла. За эту функцию внутри компонента карточки отвечает объемный метод *likeToggle()*. Код метода приведен ниже:

```
likeToggle(id) {
    this.inLiked = !this.inLiked;
    this.inLiked === true
      ? (this.likeActive = "articles__like-active")
      : (this.likeActive = "");
    if (this.inLiked) {
      this.rewriteLocalUserData({
        key: "likedProducts",
        value: [...this.inLikedList, id],
      });
      this.addCardInLiked(this.article);
    } else {
      this.rewriteLocalUserData({
        key: "likedProducts",
        value: this.inLikedList.filter((item) => item !== id),
      });
      this.delCardInLiked(this.article);
    }
  },
```

Когда пользователь нажимает на иконку сердечка карточки товара как раз и срабатывает данный метод. Он состоит из двух частей, во-первых, булевая переменная *inLiked* меняет свое значение на противоположное, а иконка сердечка меняется визуально, отображая что товар находится в избранном или удален оттуда (Рисунок 7).

Рис. 7 – Визуальное отображение товара в избранных

Во-вторых, id карточки добавляется или удаляется из локальных данных браузера с помощью мутации *rewriteLocalUserData()*, а затем сама карточка также добавляются либо удаляются из массива избранных товаров с помощью мутаций *addCardInLiked()* и *delCardInLiked()*.

```
rewriteLocalUserData(state, chahgedData) {
    const userData = JSON.parse(localStorage.getItem("userData"));
    userData[chahgedData.key] = chahgedData.value;
    localStorage.setItem("userData", JSON.stringify(userData));
    this.commit("user/changeUserData", userData);
},
```

Перезапись данных в *localStorage* происходит следующим образом. Сначала считывается объект данных пользователя из браузера, а затем изменяется определенное свойство этого объекта. После чего объект снова записывается в локальные данные браузера.

Страница результатов поиска по структуре очень схожа со страницей избранных товаров. Ее задача также состоит в том, чтобы отображать список товаров и соответственно имеет самую простую логику:

```
<script>
 import SecondCard from "@/components/SecondCard.vue";
 import { mapState } from "vuex";
 export default {
  components: {
   SecondCard,
  },
  computed: {
   ...mapState({
    searchResalt: (state) => state.searchResalt,
   }),
  },
 };
 </script>
```

По сути компонент просто импортируют массив результатов из Vuex-store. Основную же логика реализующая поиск товаров на странице расположена в компоненте HeaderComp. Основной код необходимый для поиска товаров расположен ниже:

```
methods: {
   ...mapActions({
     searchProduct: "searchProduct",
   }),
   openmenu() {
     this.menushow = !this.menushow;
     this.menushow
       ? (this.marginBottom = "margin-bottom")
       : (this.marginBottom = "");
   },
   search() {
     if (this.searchValue) {
       this.searchProduct(this.searchValue);
       this.$router.push(`/search/${this.searchValue}`);
     }
     this.searchValue = "";
     this.openmenu();
   },
}
```

Метод *search()* работает следующим образом. Если пользователь оставляет поисковую строку сайта пустой, то по клику на кнопку поиска меню поиска просто закроется. Если же пользователь ввел какую либо фразу в строке, то метод *search()* запускает *action*-метод *searchProduct()*, а затем переключает *<router-view> </router-view>* на компонент страницы результатов поиска, обнуляет в *HeaderPage* поле поиска и скрывает меню. Поиск товаров изображен на рисунке 8.

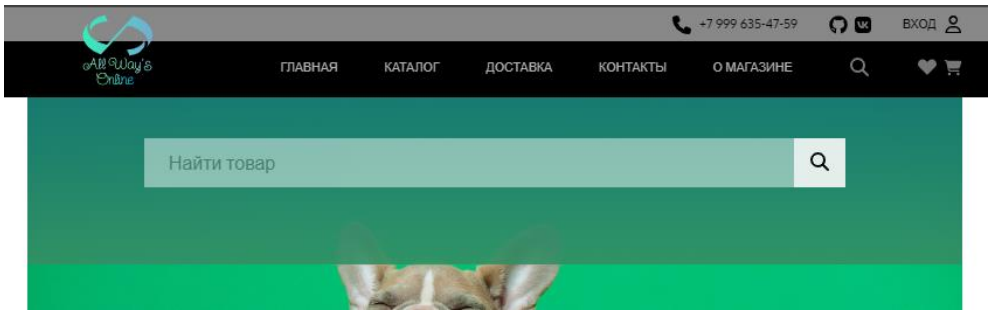


Рис.8 – Меню поиска товаров

Action-метод *searchProduct()* – это асинхронный метод который принимает строку результатов поиска и формирует соответствующий запрос через шаблонную строку.

3.4 Компонент страницы товара

Страница товара ProductPage была уже кратко рассмотрена в п.п.2.3. Давайте рассмотрим страницу товара подробнее. Основная проблема, возникшая при разработке компонента, была в том, что с виду простой функционал отображения карточки товара пришлось дополнить методами взаимодействия с *localStorage()*, т.к. было принято решение сделать функционал добавления и удаления карточки товаров из корзины. Рассмотрим логику работы страницы подробнее. Для начала на этапе создания или обновления компонент проверяет находится ли карточка в корзине, за это отвечает метод *checkInBasket()* который вызывается в хуках *created()* и *mounted()* код метода представлен ниже:

```
checkInBasket() {
    if (
      this.basketList.filter((item) => item.id === this.product.id).length
    ) {
      this.inBasket = true;
    } else {
      this.inBasket = false;
    }
},
```

В зависимости от наличия или отсутствия товара в корзине свойство *inBasket* принимает истинное либо ложное значение. Это влияет на контент кнопки, который меняется с помощью тернарного оператора внутри выражения:

```
{{ inBasket ? "Убрать" : "В корзину" }}
```

Изменение контента кнопки изображено на Рисунке 9.

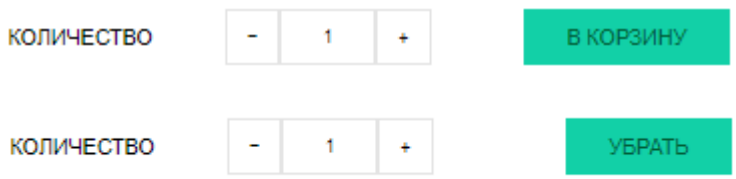


Рис. 9 – Добавление и удаление товара

Следующие два метода позволяют пользователю вручную выбрать количество товара, который будет добавлен в корзину. Код методов приведен ниже:

```
decreseQuantity() {
    this.quantity > 1 ? this.quantity-- : false;
},
increseQuantity() {
    this.quantity <= 100 ? this.quantity++ : false;
},
```

Эти две метода просто инкрементируют или декрементируют переменную *quantity* при клике по соответствующей кнопке.

Далее при нажатии на кнопку "в корзину" пользователь активируем метод переключатель *toggleInBasket()*, который взаимодействует с *localStorage* браузера и записывает либо удаляет в массив товаров в корзине объект содержащий *id* товара и *value* количество товара.

```
toggleInBasket() {
    if (!this.inBasket) {
      this.rewriteLocalUserData({
        key: "inBasketProducts",
        value: [
          ...this.basketList,
          { id: this.product.id, value: this.quantity },
        ],
      });
      this.inBasket = true;
    } else {
      this.rewriteLocalUserData({
        key: "inBasketProducts",
        value: this.basketList.filter((item) => item.id !== this.product.id),
      });
      this.inBasket = false;
    }
  },
```

Суть метода в том, что в зависимости от истинности переменной *inBasket* в мутацию *rewriteLocalUserData()* передается массив объектов либо включающий в себя текущий товар, либо исключающий его.

3.5 Компонент страницы корзины товаров.

Рассмотрим подробно страницу корзины товара. Компонент имеет широкий функционал. В корзине не только отображаются все товары, добавленные туда пользователем, также подсчитывается общая стоимость каждой позиции товара, и итоговая стоимость всех товаров. Количество товаров одной позиции в корзине можно редактировать как по единичке, так и с помощью специальной кнопки-модификатора, которая позволяет добавлять по 10 единиц товара за раз. Также компонент отслеживает факт регистрации пользователя на странице. Если клиент магазина не регистрировался в магазине, то возможность сделать заказ у него будет отсутствовать, вместо кнопки купить будет доступна лишь кнопка-ссылка на страницу входа-регистрации. Такое решение обосновано тем, что без некоторых личных данных, например, адреса, или номера телефона или почты не будет никакой возможности отправить товары в пункт назначения.

Для начала, дизайн карточки товара в корзине отличается от отображаемого на остальных страницах, и для отображения товара здесь используются другой компонент *CartProduct* и основной функционал изменения количества товара, и подсчет итога по позиции реализован именно в нем. Рассмотрим код основных методов компонента *CartProduct.*

```
deleteProduct(id) {
    this.$emit("deleteProduct", id);
    this.$emit("sentTotal", -this.total);
  },
```

Метод *deleteProduct()* просто передает *id* и *total* в компонент корзины.

```
increseQuantity(price) {
    if (this.quantity >= 90 && this.multy === 10) this.multy = 1;
    if (this.quantity >= 100) return;
    if (this.quantity <= 100) this.quantity += this.multy;
    if (this.quantity <= 100) {
      this.$emit("sentTotal", price * this.multy);
      this.$emit("changeQuantity", {
        value: this.quantity,
        id: this.product.id,
      });
    }
  },
```

```
decreseQuantity(price) {
    if (this.quantity <= 10 && this.multy === 10) this.multy = 1;
    if (this.quantity === 1) return;
    if (this.quantity > 1) this.quantity -= this.multy;
    if (this.quantity > 0) {
      this.$emit("sentTotal", -price * this.multy);
      this.$emit("changeQuantity", {
        value: this.quantity,
        id: this.product.id,
      });
    }
  },
multyQuantity() {
    this.multy === 1 ? (this.multy = 10) : (this.multy = 1);
  },
},
```

Методы *increseQuantity()* и *decreseQuantity()* увеличивают либо уменьшают количество одной позиции товара в корзине. Изменение количество товара происходит с учетом работы кнопки мультипликатора (Рисунок 10), которая вызывает метод *multyQuantity()* и переключает инкремент между 1 и 10. Внутри условных конструкций указаны ограничения не позволяющий пользователю указать количество товара меньше 1, больше 100, а также ограничения которые не позволяют мультипликатору выходить за указанные пределы. Далее если все условия соблюдены, то с помощью *$emit* методы направляют в родительский компонент корзины общую стоимость позиции товара *total* и количество товара.

Рис. 10 – Компонент карточки товара в корзине

Результатом работы методов компонента карточки является *$emit* передающие в корзину необходимые данные.

Метод потомка *deleteProduct()* вызывает в компоненте корзины две функции *deleteProductFromCart()* и *calcTotal().*

```
deleteProductFromCart(indexd, id) {
    this.products.splice(indexd, 1);
    this.rewriteLocalUserData({
      key: "inBasketProducts",
      value: this.basketList.filter((item) => item.id !== id),
    });
  },

calcTotal(value) {
    this.totalPrice += value;
  },
```

Первая удаляет карточку из массива товаров в корзине и перезаписывает новый список товаров в *localStorage* с помощью уже описанного ранее метода *rewriteLocalUserData().* Вторая корректирует итоговую стоимость заказа при любом изменении количества товара либо удалении позиции из корзины.

Также удаление товара из корзины вызывает метод *changeProducts()* который позволяет обновить страницу и удалить из дерева лишний элемент.

```
changeProducts() {
    for (const item of this.basketList) {
      this.getInBasketProduct(item.id, item.value);
    }
  },
async getInBasketProduct(id, quantity) {
    try {
      const response = await fetch(`${this.url}/products/${id}`);
      const result = await response.json();
      result.quantity = quantity;
      this.products.push(result);
    } catch (e) {
      console.log(e.message);
    }
  },
```

Метод потомка *increseQuantity()* и *decreseQuantity()* вызывают в компонента родителя две функции *rewriteQuanity()* и *calcTotal().*

```
rewriteQuanity(product) {
    this.changeInBasketProductValue(product);
    this.rewriteLocalUserData({
      key: "inBasketProducts",
      value: [...this.basketList],
    });
  },
```

Функция *rewriteQuanity()* изменяет количество товара в корзине, перезаписывая также его в *localStorage.*

И наконец если пользователь зарегистрирован, то появится кнопка, позволяющая клиенту сделать заказ. Осуществить заказ позволяют три основные функции *makeOrder(), getUserOrders()* и *getUserOrderProduct().*

```
makeOrder() {
    this.rewriteLocalUserData({
      key: "currentOrders",
      value: [
        ...this.currentOrders,
        {
          id: this.currentOrders.length,
          sum: this.totalPrice,
          status: "Проверка данных",
          products: [...this.basketList],
        },
      ],
    });
    this.rewriteLocalUserData({
      key: "inBasketProducts",
      value: [],
    });
    this.getUserOrders();
  },
```

Метод *makeOrder()* делает следующее: добавляет заказ в локальные данные браузера позволяя сохранить их при перезагрузке страницы, удаляет все товары из корзины и вызывает метод *getUserOrders().*

```
getUserOrders() {
    this.clearOrders();
    for (const order of this.currentOrders) {
      this.getUserOrderProduct(order);
    }
  },
```

Метод добавляет заказы в хранилище приложения вызывая в цикле функцию *getUserOrderProduct()* для каждого из заказов, т.к. человек может сделать несколько заказов сразу.

```
async getUserOrderProduct({ id, status, sum, products }) {
    const tempProducts = [];
    for (const item of products) {
      try {
        const response = await fetch(`${this.url}/products/${item.id}`);
        const result = await response.json();
        result.quantity = item.value;
        tempProducts.push(result);
      } catch (error) {
        console.log(error.message);
      }
    }
    this.addInOrders({ id, status, sum, products: tempProducts });
  },
```

И наконец метод *getUserOrderProduct()* загружает с сервера все товары одного заказа. Благодаря этим трем функциям заказы не только сохраняются в данных пользователя, но и также могут отображаться на странице личный кабинет клиента.

3.6 Компоненты страниц вход/регистрация и личный кабинет.

Принципы работы и взаимодействие страниц вход/регистрация и личный кабинет были уже кратко описаны в п.п. 2.3. Давайте рассмотрим подробнее как работает данная составляющая приложения.

На страницу входа изначально можно попасть двумя способами: через компонент *HeaderComp* по ссылке либо со страницы корзины, которая направляет клиента туда, если вход не был выполнен. Далее по логике, клиент должен либо произвести вход, либо зарегистрироваться на сайте. При вводе логина и пароля у пользователя есть возможность поставить или убрать галочку над чекбоксом "запомнить меня". И если в результате входа чекбокс был отмечен пользователем, то включается функция автоматической авторизации и при перезагрузке страницы снова производить вход не потребуется. В компоненте *HeaderComp* это выглядит следующим образом:

```
{{userAuth ? "Личный кабинет" : "Вход"}}
```

Если пользователь зарегистрирован, то сразу будет отображаться ссылка на личный кабинет. Код отвечающий за эти возможности приведен ниже:

```
created() {
   if (this.userData.emailLogin) {
     this.login = this.userData.emailLogin;
     this.autoSignIn();
   } else {
     this.authentication = false;
   }
 },
```

Сначала в хуке *created()* компонент проверяет зарегистрирован пользователь или нет. Если же зарегистрирован то выполняется метод *autoSignIn()*. Он устанавливает дефолтный пароль и вызывает метод *signIn()*

```
autoSignIn() {
     this.password = "Qwerty123";
     if (this.userData.remember) {
       this.signIn();
      }
    },
```

В этом методе просто сравниваются дефолтный пароль и логин пользователя с заданными заранее и производится переход на страницу личный кабинет. Правильным решением было бы передавать пароль и логин на сервер, и получая ответ уже производить вход, но JSON-сервер используемый в дипломной работе не позволяет сохранять какие-либо данные, поэтому приведенные в этом подпункте методы лишь имитируют необходимый для разработки функционал.

```
signIn() {
    const user = this.getLoginPassword();
    if (this.login === user.login && this.password === user.password) {
      this.rewriteLocalUserData({ key: "remember", value: true });
      this.$router.push("/account");
    } else {
      this.authentication = false;
    }
  },
getLoginPassword() {
    // Defalt password
    return { login: this.userData.emailLogin, password: "Qwerty123" };
  },
```

Все поля форм входа и регистрации производят валидацию введенных пользователем данных с помощью атрибута *pattern* в котором указывается регулярное выражение, описывающее правила валидации. Примеры выражений приведены ниже:

1. Для фамилии, имени либо отчества – это первая заглавная буква на русском, либо английском языке и далее 14 строчных букв.

```
pattern="(^[A-Z]{1}[a-z]{1,14}$)|(^[А-Я]{1}[а-я]{1,14}$)"
```

2. Для телефона всегда начинается с +7, а далее 10 цифр.

```
pattern="^\+7[1-9]{10}$"
```

3. Для пароля стандартные требования хотя бы одна заглавная буква, хотя бы одна строчная буква, одна цифра и длина пароля не менее 8 символов.

```
pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}"
```

4. Повтор пароля связывает атрибут со значением поля пароля, и требует равенства введенных паролей в двух полях ввода.

:pattern="password"

5. Для логина набор правил исключает дублирование некоторых символов, также не стоит забывать про встроенную валидацию в HTML.

pattern="^(?!.@.*@.*$)(?!.*@.*--.*\..*$)*
(?!.@.*-\..*$)(?!.*@.*-$)((.*)?@.+(\..{1,11})?)$"*

Рассмотрим наконец страницу "личный кабинет". Структурно компонент можно разделить на левую часть в которой находится переключатель и правую, где отображается контент. Эскизы страницы вы можете увидеть в Приложениях №7-10 дипломной работы. Работает она с помощью условного рендеринга, отображая тот контент, на который указывает состояние переключателя. За работу данной функции страницы отвечает метод *activate()*. Код метода приведен ниже:

```
activate(index, item) {
    if (index !== this.activeItem) {
      this.accountList[this.activeItem].active = "";
      item.active = "account__item-active";
      this.activeItem = index;
    }
  },
```

Каждый элемент переключателя имеет свой индекс. По клику на элемент, он меняет визуальное отображение получая активный класс через выражение: *active = "account__item-active"*, меняется также поле *activeItem* на индекс текущего активного элемента и обнуляется активный класс у прошлого элемента. При этом в зависимости от значения *activeItem* будет показан требуемый контент в правой части экрана.

Сам по себе функционал отображения контента не требуют сложной логики, нужно всего лишь импортировать набор *computed*-свойств из состояния Vuex-store. Поэтому акцентироваться на нем не стоит.

Для реализации возможности редактирования личных данных клиента введенных пользователем при регистрации, в компоненте используется методы *readInputValues()* и *saveUserData()*. Код методов приведен ниже:

```
readInputValues() {
    this.login = this.user.emailLogin;
    this.name = this.user.name;
    this.surname = this.user.surname;
    this.lastname = this.user.lastname;
    this.phone = this.user.phone;
    this.adress = this.user.adress;
  },
saveUserData() {
    const userDataChanged = {
     emailLogin: this.login,
     // password: this.password,
     name: this.name,
     surname: this.surname,
     lastname: this.lastname,
     phone: this.phone,
     adress: this.adress,
    };
    this.rewriteLocalUserObject(userDataChanged);
  },
```

Первый метод активируется на этапе создания компонента в хуке *created()* и записывает данные клиента в собственные переменные. Затем при активации режима редактирования данных пользователь может изменить их отредактировав соответствующую строку ввода данных. Поля ввода динамически связаны с одноименными переменными благодаря директиве *v-model*. Валидация полей ввода здесь такая же, как и на странице регистрации, что не позволит клиенту просто удалить данные, не указывая корректных взамен. После того как пользователь нажмет кнопу сохранить активируется метод *saveUserData()*. Как понятно из названия метода, он просто перезаписывает данные в *localStorage* браузера.

В итоге детальный разбор работы основных компонентов можно считать завершенным и успешным. Большое количество запланированных функций удалось реализовать в рамках данной дипломной работы.

ЗАКЛЮЧЕНИЕ

В рамках дипломной работы было разработано полноценное веб-приложение. Интернет-магазин был создан по принципам Single Page Application архитектуры. Во время разработки приложения были использованы инструменты VueCLI фреймворка Vue 3 версии.

Созданное веб-приложение способно получать данные карточек товаров с сервера, осуществлять фильтрацию товаров по выбранной категории, отображать сортированные по нескольким признакам товары, добавлять товары в избранное или корзину, сохранять данные в браузере на стороне клиента.

Цель дипломной работы: "Изучение фреймворка Vue 3 версии и создание полноценного SPA интернет-магазин с его помощью" можно считать достигнутой т.к. были выполнены основные задачи проекта:

1. Была изучена документация по Vue 3 версии и его библиотекам, позволяющим создавать SPA приложения;
2. Разработаны дизайн и структура всех страниц сайта;
3. Были разработаны основные асинхронные методы получения товаров с сервера;
4. Настроено взаимодействие страниц приложения между собой через глобальное хранилище данных с помощью Vuex-store;
5. Настроены переходы по страницам с помощью Vue-router, в том числе использовались динамические маршруты для некоторых страниц;
6. Создана главная страница приложения;
7. Созданы страницы каталога и избранных товаров;
8. Создана корзина товаров;
9. Разработаны методы взаимодействия приложения с локальным хранилищем данных;
10. Созданы страницы регистрации, входа и личный кабинет пользователя.

В рамках дипломной работы был получен бесценный опыт разработки на фреймворке Vue 3. Были изучены основные директивы, связывание, условный рендеринг, циклическая отрисовка(v-for), изучен жизненный цикл компонента, и его структура. Была рассмотрена структура проекта, получаемая с помощью VueCLI.

Также были рассмотрены основные требования к веб-приложению, большая часть которых была учтена при создании Интернет-магазина.

В рамках разработки дизайна, были созданы эскизы всех страниц приложения, выбраны основные стили компонентов цвета, шрифты, разработан дизайн карточки товара, и прочих интерактивных элементов приложения.

Были созданы все необходимые для работы приложения компоненты-страницы. Была настроена их взаимосвязь через библиотеки фреймворка Vue 3, Vuex-store и Vue-router. Созданы основные переменные состояния, в которых приложение хранит основные данные для работы и взаимодействия. Разработаны методы управления состоянием – мутации. Созданы методы получения данных с сервера. Выделен отдельный модуль взаимодействия приложения с localStorage браузера. Настроен роутинг по компонентам преобразуя приложение в полноценное SPA. Реализованы динамические адреса страниц каталога, и каждого из товаров.

В конечном итоге, дипломный проект позволил овладеть некоторыми современными инструментами разработки, изучить основные особенности фрейимворка Vue 3, изучить важные библиотеки для разработки SPA, и наконец разработать сам Интернет-магазин.

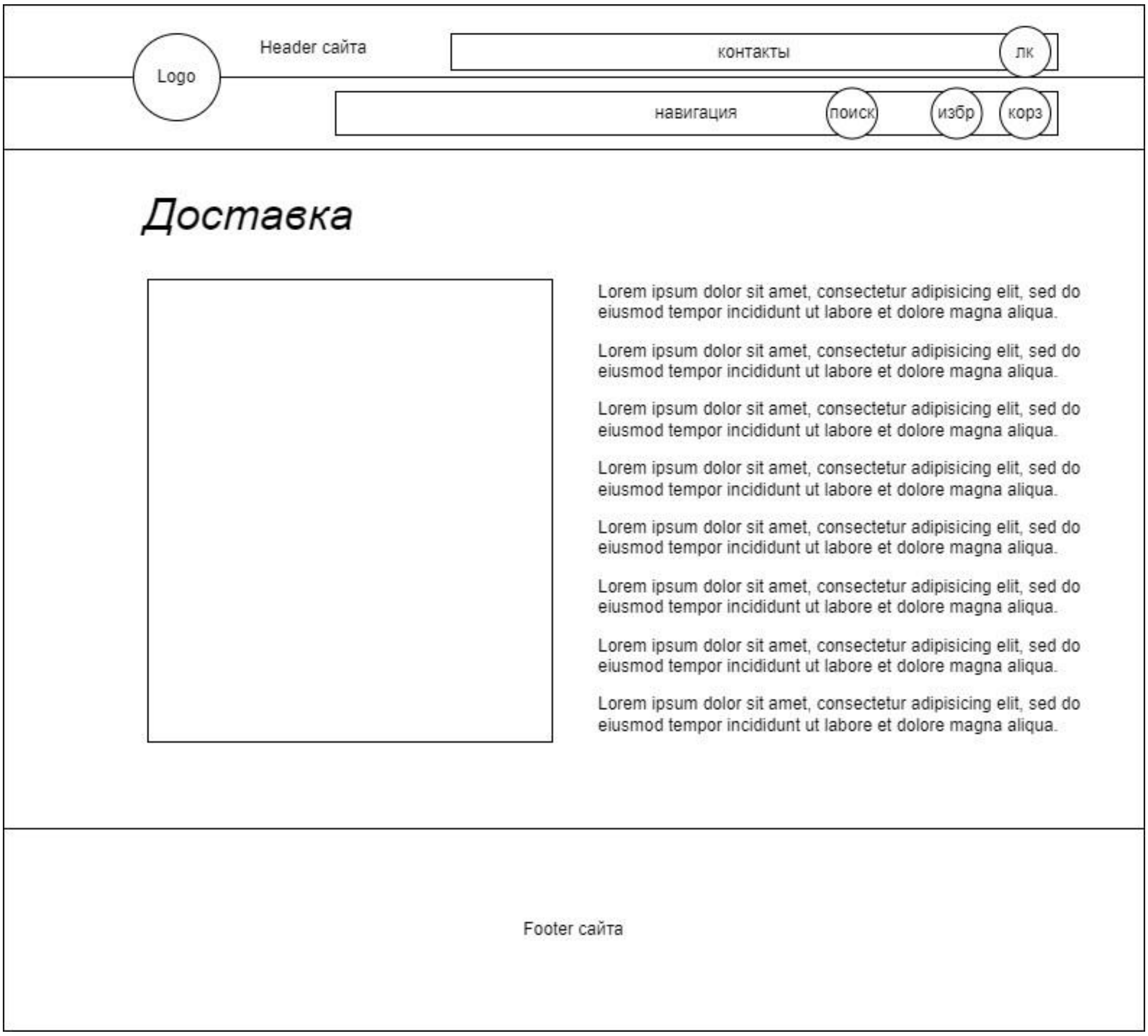
Список используемых источников

1. https://ru.hexlet.io/blog/posts/arhitektura-prilozheniya
2. https://itanddigital.ru/webapplications
3. https://habr.com/ru/companies/alfa/articles/725626
4. https://optimalgroup.ru/blog/sozdanie-veb-prilozhenij-spa-i-pwa
5. https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/HTML_basics
6. https://developer.mozilla.org/ru/docs/Learn/CSS/First_steps/What_is_CSS
7. https://developer.mozilla.org/ru/docs/Learn/CSS/Building_blocks/Cascade_and_inheritance
8. https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity#selector_weight_categories
9. https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Introduction
10. https://learn.javascript.ru/intro
11. https://v3.ru.vuejs.org/ru/guide/introduction.html
12. https://metanit.com/web/vue/1.1.php
13. Конспект лекций VueCLI
14. https://v3.ru.vuejs.org/ru/guide/single-file-component.html
15. https://netology.ru/blog/06-2022-ux-ui-designer

ПРИЛОЖЕНИЯ

Приложение 1 – Эскиз страницы каталог

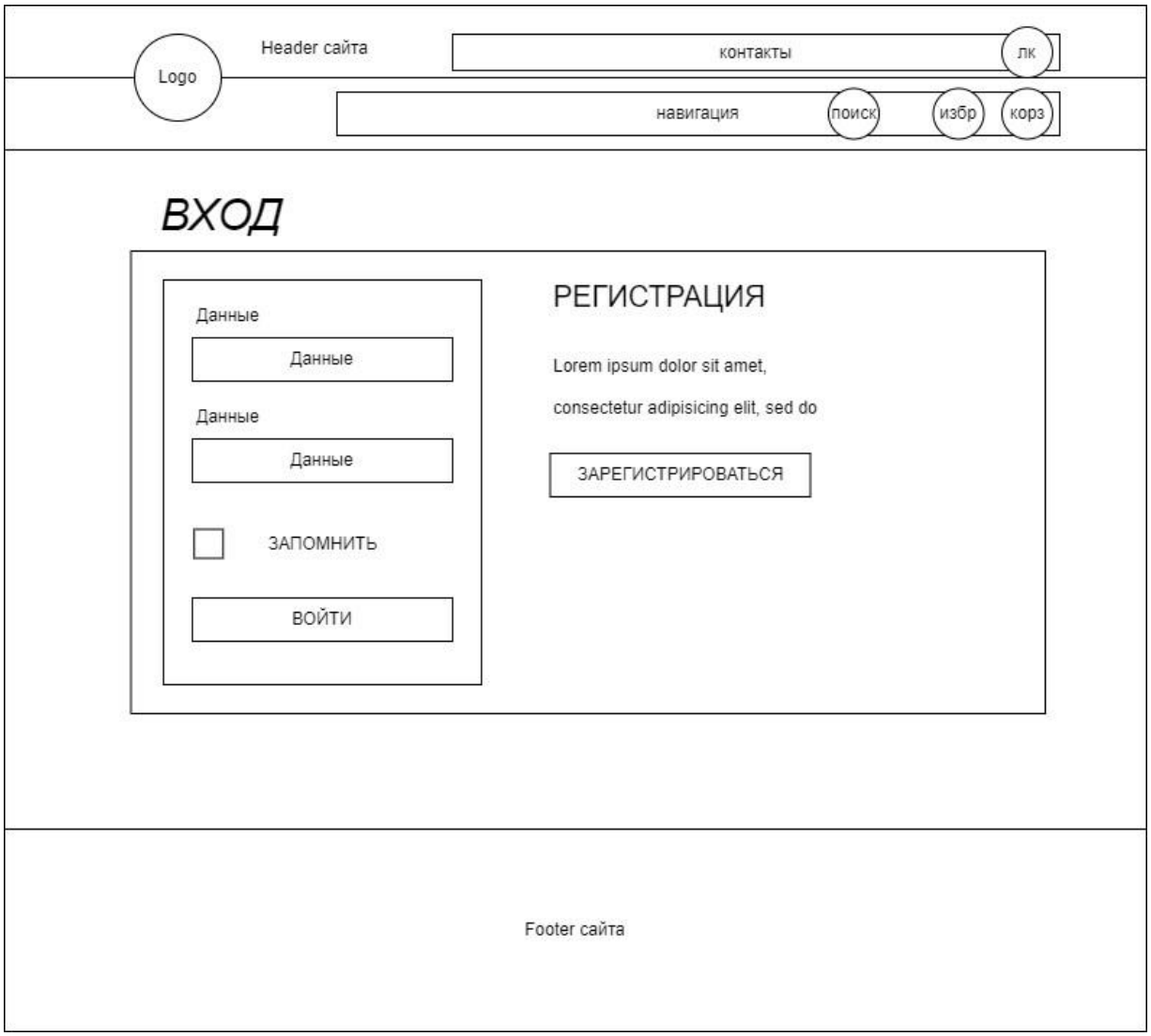
Приложение 2 – Эскиз страницы доставка



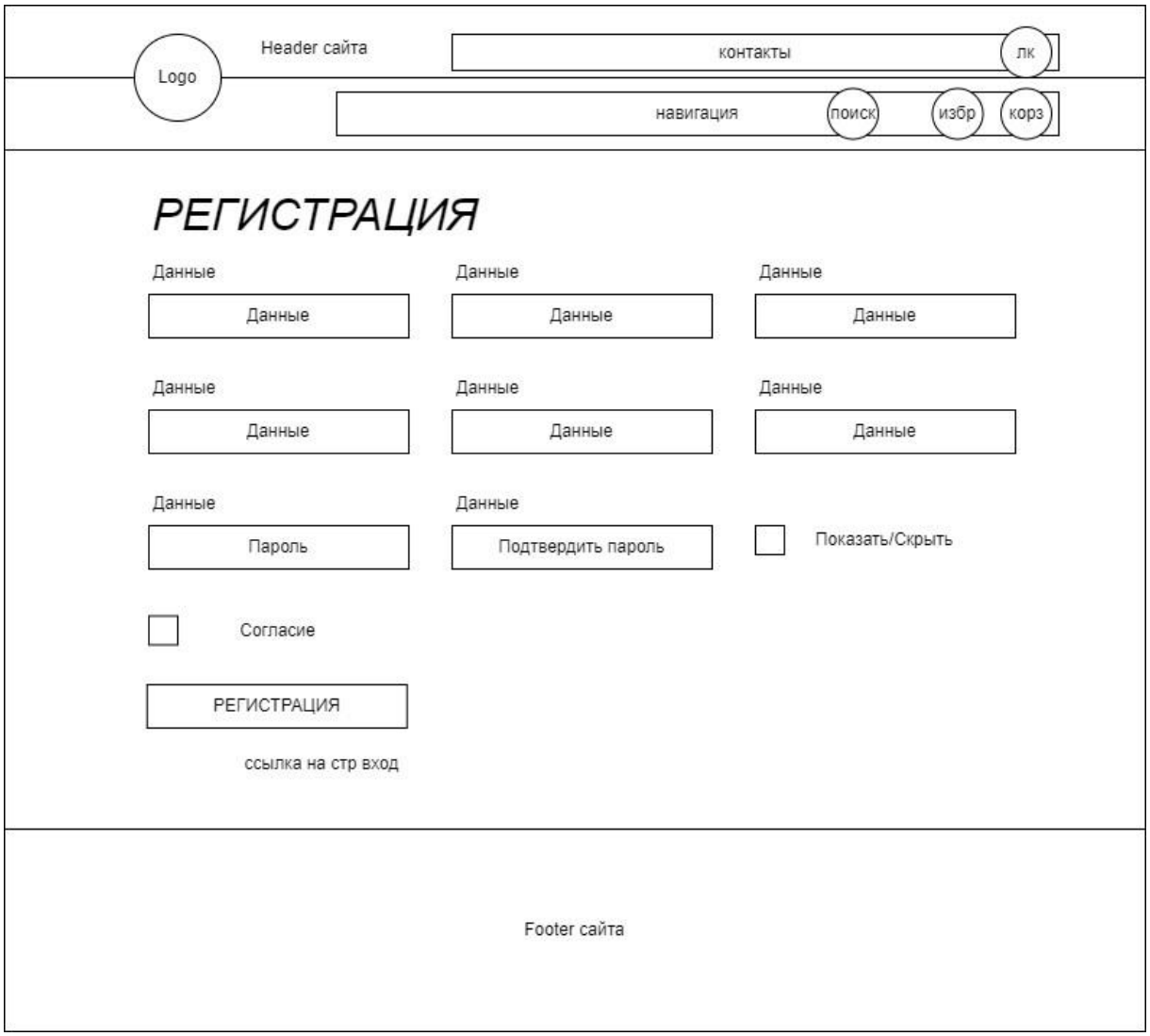
Приложение 3 – Эскиз страница контакты

Приложение 4 – Эскиз страницы о магазине

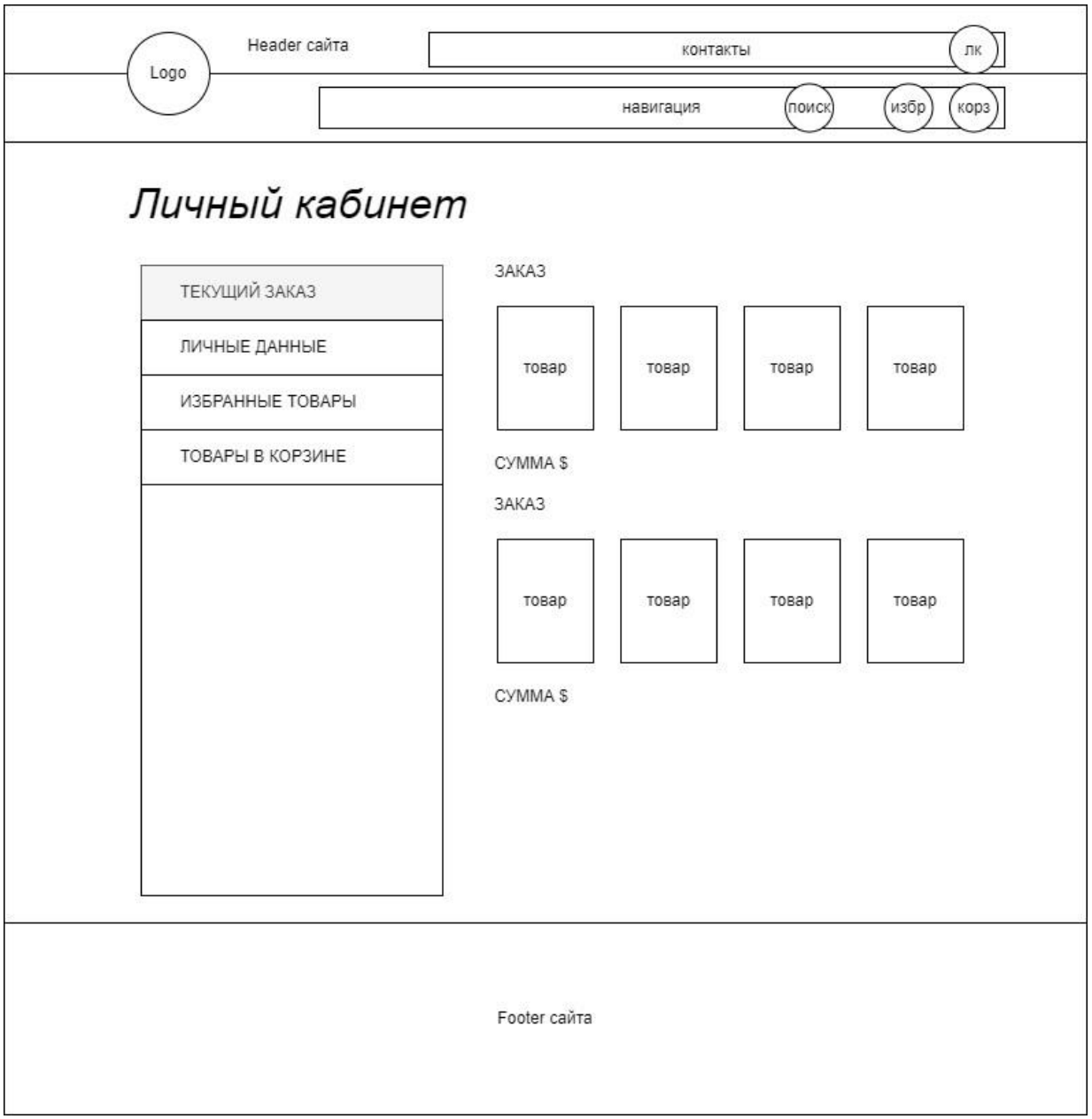
Приложение 5 – Эскиз страницы вход



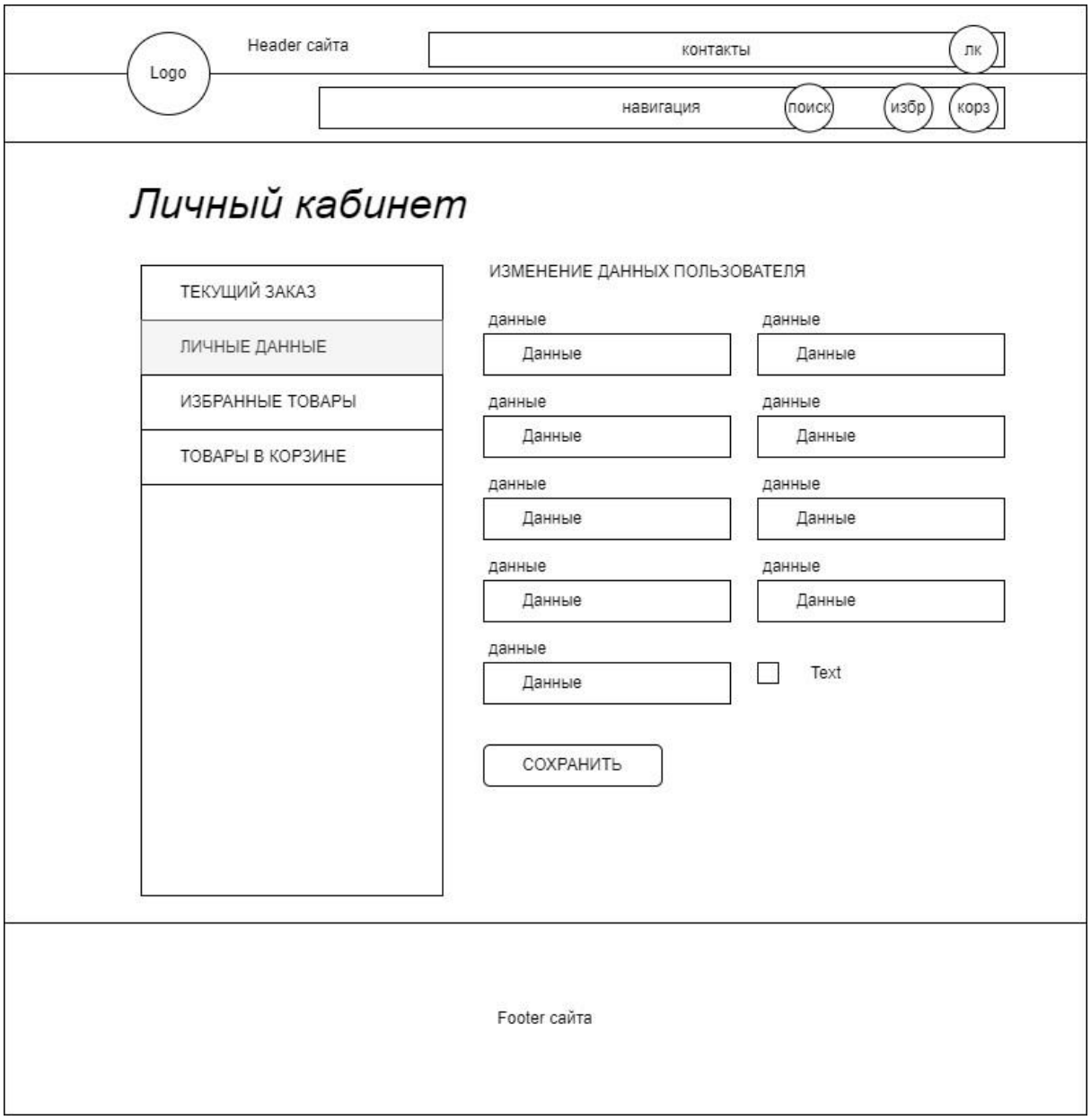
Приложение 6 – Эскиз страницы регистрация



Приложение 7 – Эскиз страницы личный кабинет – текущий заказ



Приложение 8 – Эскиз страницы личный кабинет – личные данные



Приложение 9 – Эскиз страницы личный кабинет – избранные товары

Приложение 10 – Эскиз страницы личный кабинет – товары в корзине

Приложение 11 – Эскиз страницы корзина